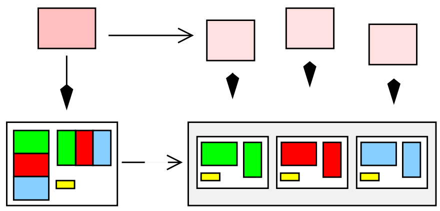
ADAPTOR
Programmers Guide
Version 6.0 (June 1998)

T. Brandes



Report ADAPTOR 3

June 24, 1998

GMD – German National Research Center for Information Technology
Institute for Algorithms and Scientific Computing, SCAI
Schloss Birlinghoven
D-53754 Sankt Augustin
Germany

Tel.: +49 (0)2241 / 14-2492
E-mail: brandes@gmd.de

This page is left empty

ADAPTOR
Programmers Guide
Version 6.0 (June 1998)

T. Brandes

GMD – German National Research Center for Information Technology
Schloss Birlinghoven, D-53754 Sankt Augustin, Germany

Contents

Abstract

ADAPTOR (Automatic DAta Parallelism TranslatOR) is a public domain High Performance Fortran (HPF) compilation system developed at the GMD during the last years. The tool transforms data parallel programs written in Fortran with array extensions, parallel loops, and layout directives into programs with explicit message passing. It can also compile the data parallel programs for virtual shared memory systems to take advantage of global address spaces. This version is especially devoted to implement the approved extensions of the HPF 2.0 standard not implemented so far. This includes especially the advanced mapping features and task parallelism. From the previous versions it contains already a lot of optimization techniques.

This manual describes the language features which are supported by ADAPTOR and where there are still some restrictions. Furthermore it will be explained how to use ADAPTOR to get efficient HPF programs.

1 Overview

The ADAPTOR tool offers the possibility to write efficient data parallel programs without explicit message passing. This is realized by using the inherent parallelism of array operations and/or parallel loops on arrays where the arrays are distributed among the available processors. Necessary communication will be generated automatically.

Attention: ADAPTOR has no features for detecting loop parallelism of a sequential loop. Sequential loops will not be parallelized automatically.

As the tool has been designed originally to run Connection Machine Fortran programs [Thi91] on MIMD architectures, the source language was strongly related to CM Fortran. CM Fortran supported already many features of Fortran 90 [ABM+92]. With the introduction of High Performance Fortran (HPF) [Hig94, KLS+94], ADAPTOR supports now this standardized data parallel language.

This manual describes which features of the different languages (FORTRAN 77, Fortran 90, Fortran 95, High Performance Fortran: Base Language and Approved Extensions) are supported.

While section 2 describes only the new features of the latest version, section 3 gives a summary of all features supported or not in ADAPTOR. Afterwards, the first part of this manual presents all the features of Fortran 90 (17.2) of HPF regarding mapping of data (4), data parallelism (7), intrinsic (17.22) and extrinsic procedures (9).

The second part of this manual is related to the execution of HPF programs on parallel machines and should give a more detailed understanding how an HPF compiler like ADAPTOR translates the data parallel programs. After a more general introduction (11) it will be explained where no communication is necessary (12), where global communication (13), structured communication (14) or unstructured communication (16) is generated by the compiler.

The appendix summarizes the new features of Fortran 90 (Section 17.2).

2 About the new Version 6.0

The main differences between version 6.0 and the previous version of ADAPTOR (version 5.1) are:

- ADAPTOR supports block-cyclic distributions, `CYCLIC(N)` with $N \geq 2$.
- ADAPTOR supports `INDIRECT` distributions as specified in the HPF 2.0 standard.
- ADAPTOR supports `ARBIRTRARY` distributions as specified by Moreira et. al. [MEKN96].

- Arrays can now be distributed onto processor subgroups.
- Processor subgroups can be used within the `ON` directive.
- ADAPTOR supports task parallelism as specified in the HPF 2.0 standard. Communication between tasks might be possible by using the array descriptors of non-local arrays (non-portable) or by using the `HPF_TASK_LIBRARY`.
- Embedded dimensions for an alignment can be used without restrictions.
- ADAPTOR supports the new machine-specific `SELECT` directive to select dimensions of arrays for vectorization and/or shared memory parallelization.

Furthermore, unstructured communications based on indirect addressing are realized more efficiently.

The ADAPTOR specific `MAP` directive (available until version 5.1) is no longer supported.

3 Overview of the ADAPTOR Input Language

The front end has been designed in such a way that all features of FORTRAN 77, Fortran 90, Fortran 95 and HPF are parsed.

Nevertheless ADAPTOR does still not support all features of these languages. But the user will get information about the unsupported features in his code.

3.1 Supported Features of Fortran 90

The following extensions of Fortran 90 [ABM+92] can be used within ADAPTOR:

- array expressions and array assignments,
- intrinsic functions for arrays (with some exceptions),
- dynamic arrays,
- array-valued functions,
- assumed-shaped arrays,
- optional arguments,
- new declaration statements,
- new loop constructs `CASE`, `EXIT`, and `CYCLE`,
- the binary operations `<>`, `/=`, `==`, `<=`, `<`, `>`, and `>=` instead of `.ne.`, `.eq.`, `.le.`, `.lt.`, `.gt.`, and `.ge.`,
- ending comments starting with `!`,
- semicolon `;` for separating statements,
- using `&` for continuation lines,
- free source format.

Furthermore, the following Fortran 90 features can be used if there is a Fortran 90 compiler available on the target machine:

- parameterized data types and numeric inquiry functions,
- modules,
- derived data types,
- contained procedures,
- generic procedures,
- overloading of operators.

3.2 Supported Features of Fortran 95

ADAPTOR supports the following features of Fortran 95 that have been formerly part of HPF 1.1

- `FORALL` statement and `FORALL` construct,
- `PURE` procedures.

3.2.1 Supported Features of HPF 2.0 Base Language

The following HPF features are supported:

- processor directives and different abstract processor arrays,
- distribution directives of HPF,
- alignment directives of HPF
- some new HPF intrinsic functions (e.g. `NUMBER_OF_PROCESSORS`, `xxx_SCATTER`),
- serial routines (`HPF_SERIAL`),
- independent `DO` loops.
- `REDUCTION` directive for `INDEPENDENT` loops,
- local and serial `EXTRINSIC` procedures.

The most important restrictions are:

- no strides in alignments to distributed dimensions that are not SERIAL or BLOCK distributed.
- no access to non-local data within `PURE` routines.

Attention: In some situations, especially when using complex independent loops, complex array expressions and complex indirect addressing, ADAPTOR will fail to translate the data parallel code. In this case, the system will give an appropriate error message and the user has to find a workaround. Experiences with previous versions have shown that this also might help to write more efficient parallel programs. Indeed, the restrictions are mainly due to the fact that in this situation ADAPTOR does not know how to generate efficient code.

3.3 Support of Approved Extensions of HPF 2.0

ADAPTOR supports the following approved extensions of HPF 2.0 [Hig97]:

- general block distributions,
- indirect distributions,
- `ON HOME` directive,
- `RESIDENT` directive.
- `TASK_REGION` construct.
- shadow edges for arrays can be explicitly defined by directives.
- processor subgroups,

Not supported are:

- pointers to distributed arrays,
- mapping of any components within derived types,

3.4 New Features of ADAPTOR

Furthermore, ADAPTOR realizes some features that are not standardized until now:

- `ARBITRARY` distributions where blocks of different sizes within one dimension can be mapped to different processors.
- ADAPTOR allows the use of the *REDUCTION* directive outside of parallel loops.
- With the `SHARED` directive, distributed arrays will be put in a shared or virtually shared memory segment (but only if this is supported on the target machine).
- Indirect addressing requires the complex computation of a communication schedule. This schedule can be reused if the involved integer arrays have not been modified. But whether an array has been changed or not requires complex data flow analysis that is not available in ADAPTOR. With the `TRACE` directive the user specifies that an integer array used for indirect addressing will be marked as invalid after an update. As long as it has not been updated, the schedule for the indirect addressing can be reused.
- ADAPTOR supports the new machine-specific `SELECT` directive to select dimensions of arrays for vectorization and/or shared memory parallelization.

3.5 Front End Problems

Currently, the front end of ADAPTOR has some problems. In the following all known problems are listed by an example that will not work in the current release.

- ADAPTOR cannot deal with Holerith constants (except in FORMAT statements, but there might also be problems in certain situations).

```
    DATA PGNAME/8HPAMCRASH/          ! synax error
450 FORMAT(1H ,20(1H!),16H FEHLERAUSGANG (,I2,1H),16(1H!))
```

- Do not use array variables with a name that has some other meaning in Fortran. In this case, the array variable should be renamed.

```
INTEGER DATA (100)
DATA = 10
DATA (50) = 3    ! this makes problems
```

- Blanks are significant for the frontend. Do not use blanks within names, numbers and relational operators.

```
C O M M O N / / A
A = 21 000 000.d0
IF (0. EQ. A) ...
```

- no `PAUSE` statement,
- no alternate returns.

3.6 Known Problems

3.6.1 Problems with Dynamic Arrays

When translating array operations to serial `DO` loops ADAPTOR needs information about the bounds of the array dimensions. Therefore it takes the expressions of the `ALLOCATE` statement or of the array declaration.

```
      subroutine S (N)
      real, dimension (N) :: A
      real, dimension (:), allocatable :: B
      allocate (B(N))
      ...
c     A(2:) = B(:N-1) will be translated to
      do I = 2, N
         A(I) = B(I-1)
      end do
      deallocate (B)
```

Therefore the variables within the index expressions must not be redefined during the lifetime of the corresponding array.

```
      allocate (B(N))
      N = N + 1          ! might cause serious problems
      B(:) = B(:) + 1
      deallocate (B)
```

3.7 Restrictions for Redistributions

ADAPTOR supports the `REDISTRIUBTE` statement and `REALIGN` statement. This has been proven to be very convenient for some applications, especially when the distribution is computed during the initialization.

But it assumes that every access to a distributed array has only one reaching distribution, and this is the last syntactical `DISTRIBUTE` or `REDISTRIBUTE` statement.

3.8 Restrictions for the ON Directive

The ON HOME directive can restrict the execution to a single processor in one dimension, but not to a subset of processors.

```
      real, dimension (N) :: A
!hpf$ distribute (block)  :: A
      ...
!hpf$ on home (A(1))
         <statement>       ! will be executed on the first processor
!hpf$ on home (A(1:4))
         <statement>       ! will be executed on all processors
```

3.9 Random Numbers

The intrinsic routine RANDOM_DATA returns random numbers. In case of distributed arrays, this is a parallel random number generator. The penalty is that the routine delivers different random numbers when executed on different number of processors. Only when the program is executed on the same number of processors, it delivers the same results.

4 Mapping of Data

4.1 Overview of Data Mapping

The central idea of the data-parallel programming model is the mapping of the array elements and the corresponding work on these arrays onto the processors.

High Performance Fortran provides a two-level mapping of data objects to memory regions, referred to as "abstract processors". Data objects (typically array elements) are first aligned relative to one another. This group of arrays is then distributed onto a rectilinear arrangement of abstract processors. Furthermore, there is a machine-dependent mapping of abstract processors to physical processors. Figure 1 illustrates the model.

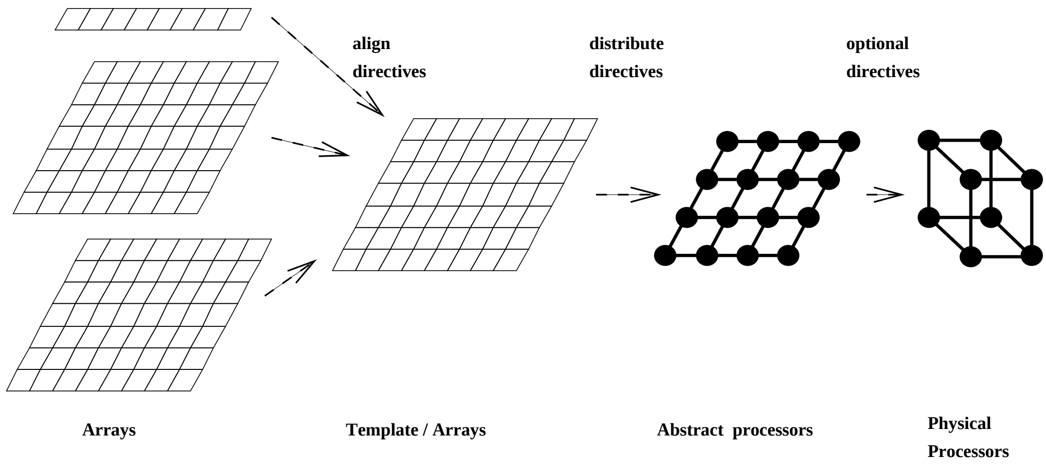


Figure 1: Directives of High Performance Fortran

Within ADAPTOR the mapping of arrays is handled in two ways:

- For the usual message passing execution model the directives specify how the data is distributed onto the processors. Every processor will only allocate memory for its local part. Global addresses have to be translated to local addresses. Sequence association between the elements of a distributed array does not exist. Furthermore, the load of parallel loops is distributed in such a way that most operations can be executed on local data.
- For shared arrays (see also section 4.6 the directives specify only how the operations on this data are distributed. The data itself reside in a shared memory segment, global addresses remain global, sequence association between the array elements still exists.
- Sequence association can be guaranteed by using the `SEQUENCE` directive (see also section 4.8). In this case, the data will be replicated on all processors.

4.2 Distribution of Arrays

A distribute directive partitions an object between processors. The user can specify the dimensions of the object (array or template) which should be distributed. A distribution guarantees that one processor owns only a part of the original array. A node processor needs only memory for this local part.

4.2.1 One-Dimensional Distributions

A one-dimensional array with N elements can be distributed onto P processors in the following way:

- block-distributed (see figure 2) where one processor contains a contiguous piece of size N/P or of an explicitly specified size B.

```
      real, dimension (N) :: A
!hpf$ distribute A(block)
!hpf$ distribute A(block(B))
```

- cyclic distributed (see figure 3) where every Pth element is on the same processor or every Pth block of size B is on the same processor.

```
      real, dimension (N) :: A
!hpf$ distribute A(cyclic)
!hpf$ distribute A(cyclic(B))
```

- general block distributed, With the usual block distribution, every processor gets the same block size. The general block distribution allows to specify a certain size for every processor.

```
      integer, dimension(4) :: SIZE = (/10,16,16,10/)
      real, dimension (52)  :: A
!hpf$ distribute A(gen_block(SIZE))
```

- indirectly distributed, where the actual mapping to the processors is specified by an integer array (map array).

```
      integer, dimension(52) :: MAP = (/1,3,4,1,...,3,2,4,1/)
      real, dimension (52)   :: A
!hpf$ distribute A(indirect(MAP))
```

- arbitrarily distributed, where the actual mapping is also specified by a map array, but instead for single elements one entry is given for a chunk of elements (see Figure 4).

  It is specified as `arbitrary (N,LENGTH,MAP)` where `N` is an integer scalar and `LENGTH` and `MAP` are integer vectors of length (at least) `N`. It causes the axis to be divided into `N` blocks, numbered $1, ..., N$. The length of block i is $LENGTH(i)$ and it is mapped to processor $MAP(i), 1 \leq MAP(i) \leq P$. The length of the N blocks must sum up to length of the axis.

```
      integer, dimension (6) :: LENGTH = (/10,8,8,12,6,6/)
      integer, dimension (6) :: MAP = (/1,2,3,2,3,1/)
      real, dimension (52)  :: A
!hpf$ distribute A (arbitrary(6,LENGTH,MAP))
```

P_1	P_2	P_3	P_4
1	14	27	40
2	15	28	41
3	16	29	42
4	17	30	43
5	18	31	44
6	19	32	45
7	20	33	46
8	21	34	47
9	22	35	48
10	23	36	49
11	24	37	50
12	25	38	51
13	26	39	52

(a) DISTRIBUTE A (BLOCK)

P_1	P_2	P_3	P_4
1	15	29	43
2	16	30	44
3	17	31	45
4	18	32	46
5	19	33	47
6	20	34	48
7	21	35	49
8	22	36	50
9	23	37	51
10	24	38	52
11	25	39	-
12	26	40	-
13	27	41	-
14	28	42	-

(b) DISTRIBUTE A (BLOCK(14))

Figure 2: Block distribution of a one-dimensional array onto 4 processors

When a dimension is distributed, only the local part of it will be allocated on a single processor unless a shadow edge is specified. General block and indirect distributions are approved extensions of HPF 2.0. Arbitrary distributions have been used by Moreira et al. [MEKN96], they are not standardized, but can be considered as a mixture of general block and indirect distributions.

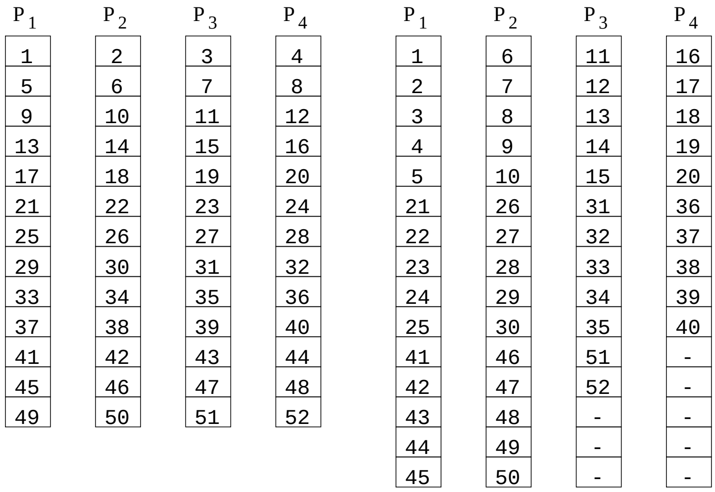
4.2.2 Multi-Dimensional Distributions

Multi-dimensional arrays can also be distributed on one-dimensional processor arrays. Only one dimension of the array will be distributed while the other dimensions become serial (see Figure 5).

In case of multi-dimensional arrays it is possible to distribute them onto multi-dimensional processor arrays. For the distributed dimensions, the user can specify different distributions (see Figure 6).

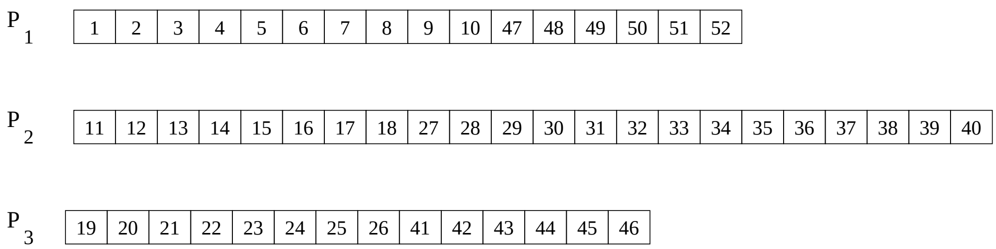
In the current version of ADAPTOR up to four dimensions can be distributed.

```
      real, dimension (N,N,N,N) :: C
!hpf$ distribute C(block, block, *, cyclic)
```



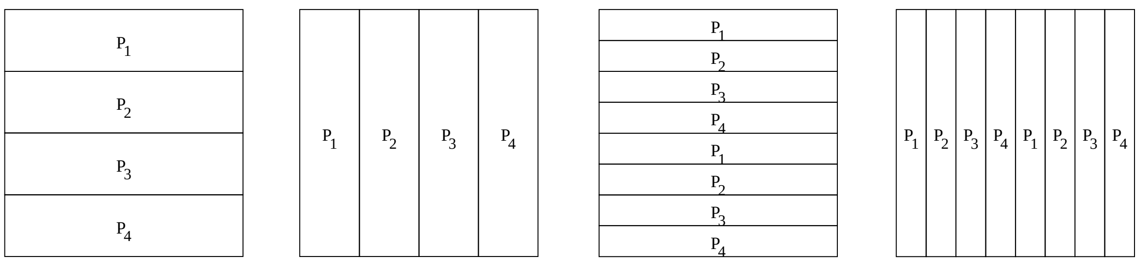
(a) DISTRIBUTE A (CYCLIC) (b) DISTRIBUTE A (CYCLIC(5))

Figure 3: Cyclic distribution of a one-dimensional array onto 4 processors



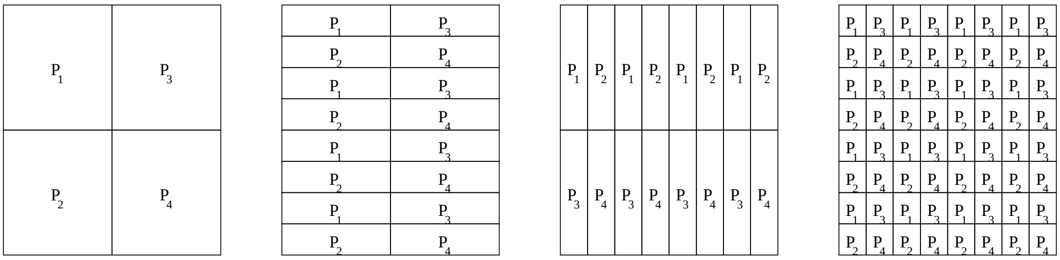
DISTRIBUTE A (ARBITRARY (6, LENGTH = [10,8,8,12,6,6], MAP = [1,2,3,2,3,1])

Figure 4: Arbitrary distribution of a one-dimensional array onto 3 processors



(a) DISTRIBUTE A(BLOCK,*) (b) DISTRIBUTE A(*,BLOCK) (c) DISTRIBUTE A(CYCLIC,*) (d) DISTRIBUTE A(*,CYCLIC)

Figure 5: Distributions of a two-dimensional array with a serial dimension



(a) DISTRIBUTE A(BLOCK,BLOCK) (b) DISTRIBUTE A(CYCLIC,BLOCK) (c) DISTRIBUTE A(BLOCK,CYCLIC) (d) DISTRIBUTE A(CYCLIC,CYCL

Figure 6: Distributions of a two-dimensional array onto 4 processors.

4.3 Abstract Processor Arrays

Arrays are distributed onto a rectilinear arrangement of abstract processors (also called *torus*). The following commands will define a one-dimensional torus with 4 processors, one with 8 processors and a two-dimensional torus with 4 processors.

```
!hpf$ processors P1 (4)
!hpf$ processors P2 (8)
!hpf$ processors P3 (2,2)
```

The definition of processor arrays is like the definition of automatic arrays (see section 17.4.2). The size can be specified with values that will only be known at runtime.

```
      subroutine SUB (N1, N2)
      integer N1, N2
!hpf$ processors TORUS1 (N1, N2)
!hpf$ processors TORUS2 (number_of_processors()/N1,N1)
```

Arrays can be distributed onto such processor arrays.

```
      real, dimension (M1,N1) :: A1
      real, dimension (M2,N2) :: A2
      real, dimension (M3,N3) :: A3
!hpf$ distribute A1 (*,block) onto P1
!hpf$ distribute A2 (*,block) onto P2
!hpf$ distribute A3 (cyclic,block) onto P3
```

Operations on A1 will only be executed by processors that belong to the abstract processor array P1.

Arrays can also be mapped to properly specified processor subsets. This is most useful in conjunction with the tasking construct, see Section 8.1.

```
!hpf$ processors P (10)
      real, dimension (N, N) :: A1, A2
!hpf$ distribute A1 (*,block) onto P(1:5)
!hpf$ distribute A2 (*,block) onto P(6:10)
```

4.4 Alignment of Arrays

ADAPTOR supports the alignment directives of HPF. In the following some examples of using alignment directives are given:

4.4.1 Alignment to a Template

The first example shows a typical use of a template. If the distribution of the template is changed (only one declaration), the distribution of all aligned arrays will be changed.

```
!hpf$ template T (NQPX,NQPX,NKPX)
!hpf$ distribute T (*,*,block)

      double precision, dimension (5,NQPX,NQPX,NKPX) :: UO, U, UP
      double precision, dimension (5,NQPX,NQPX,NKPX) :: FU, GU, HU, KU
      double precision, dimension   (NQPX,NQPX,NKPX) :: TMP, TMP1

!hpf$ align (*,:,:,:) with t(:,:,:) :: uo, u, up
!hpf$ align (*,i,j,k) with t(i,j,k) :: fu, gu, hu, ku
!hpf$ align (:,:,:) with t(:,:,:) :: tmp
!hpf$ align with t :: tmp1
```

It is recommended to use align-dummies in the alignment directive. The use of ":" causes problems for different shapes.

```
      PARAMETER (n=100)
      REAL a(1:n), b(2:n+1)
!hpf$ distribute a(block)
!hpf$ align b(i) with a(i-1)
C     the directive align b(:) with a(:) results in wrong code
```

4.4.2 Serial Dimensions

Looking at the source of the alignment, a source dimension can be mapped to a dimension of the target or not. In the latter case, the dimension of the source array is collapsed, it becomes a serial dimension.

```
!HPF$ TEMPLATE T(N)
      REAL A(N,N), B(N,N)
!HPF$ ALIGN A(I,J) WITH T(J)
!HPF$ ALIGN B(I,J) WITH T(I)
```

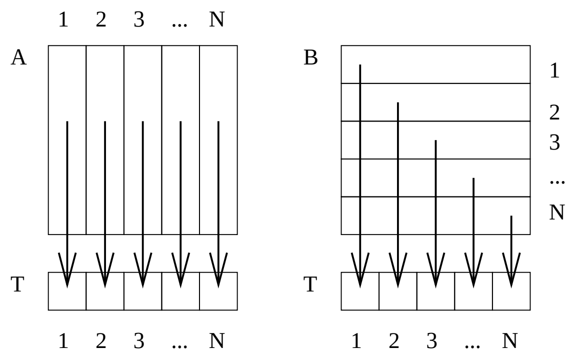


Figure 7: Serial/collapsed dimensions.

A collapsed dimension is the same as aligning it to a serial template dimension.

```
!hpf$ template T (N)              !hpf$ template T(N,N)
      REAL A(N,N)                       REAL A(N,N)
!hpf$ align A(*,J) with T(J)      !hpf$ align A(I,J) with T(I,J)
!hpf$ distribute T(block)         !hpf$ distribute T(*,block)
```

Serial dimensions are very important as the compiler will already know at compile time that no communication will be involved if the accesses have only different values in the serial dimensions.

```
!hpf$ template T (N)
      REAL A(N,N), B(N,N)
!hpf$ align A(*,I) with T(I)
!hpf$ align B(I,*) with T(I)

      A(I,K) = A(J,K)    ! no communication
      B(I,K) = B(I,J)    ! no communication
      B(1,3) = A(5,1)    ! no communication
```

4.4.3 Permutations

The HPF alignment directive allows also to interchange dimensions between alignee and align-target.

```
!hpf$ template T3 (N,N,N)
      REAL A3(N,N,N)                       ! permutation of dimensions
!hpf$ align A3(I,J,K) with T3(K,I,J)
```

4.4.4 Linear Embeddings

An alignment represents an embedding if the specified mapping is an injective mapping.

The embedding can be given by an injective mapping for every dimension.

```
!hpf$ template T2 (N,N)
      REAL A2 (N2,N2)                      ! embedding
!hpf$ align A2(I,J) with T2(2*I,2*J-1)
```

```
!hpf$ template T2 (0:N+1,0:N+1)
      REAL A2 (N,N)                      ! embedding
!hpf$ align A2(I,J) with T2(I,J)
```

```
!HPF$ TEMPLATE T(0:N+1)
      REAL X1(N)
      REAL X2(N)
!HPF$ ALIGN X1(I) WITH T(I-1)
!HPF$ ALIGN X2(I) WITH T(I+1)
```

```
!HPF$ TEMPLATE T(1:2*N)
      REAL X1(N)
      REAL X2(N)
!HPF$ ALIGN X1(I) WITH T(2*I)
!HPF$ ALIGN X2(I) WITH T(2*I-1)
```

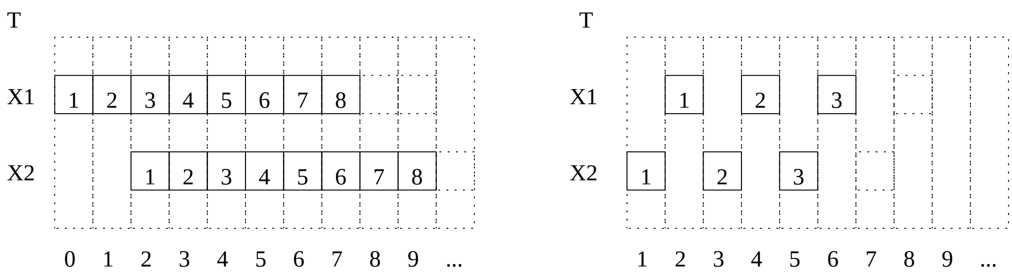


Figure 8: Linear embedding of an array into a template.

4.4.5 Embedded Arrays

If the alignee has less dimensions than the align-target, the array can be mapped to certain positions in the align-target.

Embedded dimensions are currently not supported with ADAPTOR.

```
!HPF$ TEMPLATE T(N,N)
      REAL A(N), B(N)
!HPF$ ALIGN A(I) WITH T(I,3)
!HPF$ ALIGN B(I) WITH T(3,I)
```

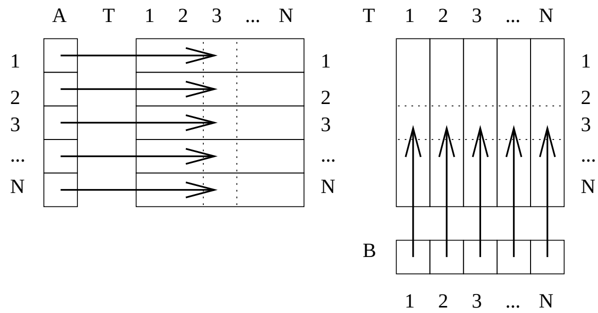


Figure 9: Embedding of an array into a template.

```
      PARAMETER (n=100,m=50)
      REAL a(m,n)
      REAL t1(m), tn(m)
!hpf$ distribute a(block,block)
!hpf$ align t1(i) with a(i,1)
!hpf$ align tn(i) with a(i,n)
      ...
      tn(1:m) = a(1:m,n)       ! no communication
      a(1:m,1) = t1(1:m)       ! no communication
```

4.4.6 Replication of Arrays

```
!HPF$ TEMPLATE T(N,N)
      REAL A(N), B(N)
!HPF$ ALIGN A(I) WITH T(I,*)
!HPF$ ALIGN B(I) WITH T(*,I)
```

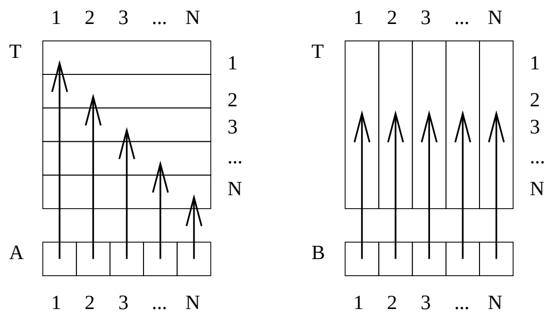


Figure 10: Replication of an array

```
!hpf$ template T2 (N,N)
      REAL A1 (N)                        ! replication along one dimension
!hpf$ align A1(I) with T2(I,*)
```

4.4.7 Restrictions for Alignment

Currently, ADAPTOR has the following restrictions for the alignment:

- ADAPTOR must know the values for a linear embedding at compile time.

```
      REAL A(M), B(N)
!hpf$ align A(I) with B(C*I+D)    !  C, D must be known at compile time
```

- Scalar variables cannot be aligned.

```
      REAL A, B(N)
!hpf$ align A with B(5)   ! no alignment for scalar variables
```

- Alignment with a stride is only supported to a serial dimension or to a block distributed dimension.

```
      real, dimension (N,N) :: A
      real, dimension (M,M) :: B
!hpf$ distribute A(block,cyclic(3))
!hpf$ align B (I,J) with A(I,J)        ! okay
!hpf$ align B (I,J) with A(2*I,J)      ! okay
!hpf$ align B (I,J) with A(I,2*J)      ! not allowed
```

4.5 DYNAMIC Directive

`ALIGN` and `DISTRIBUTE` directives are declarative directives and must be placed in the declaration part of a unit. They define a *static* data mapping: once declared it cannot be changed at execution time.

HPF allows also for dynamic mapping of arrays. The executable directives `REALIGN` and `REDISTRIBUTE` allow dynamic redistributions. These directives have the same syntax as the declarative directives, respectively, and they must be placed in the execution part.

In addition, data objects that should be remapped dynamically must be declared as dynamic using the `DYNAMIC` directive.

```
      REAL A(N,N)
!hpf$ DYNAMIC :: A
!hpf$ distribute A(block,*) :: A
      ...
!hpf$ REDISTRIBUTE A(*,block) :: A
      ...
```

4.6 Shared Arrays

ADAPTOR supports shared arrays that will be mapped to the global address space of a parallel machine.

Attention: This feature is only available if the MIMD system supports something like shared data structures or a global address space. Currently the most known feature are the System V shared segments.

```
      real A(6,8), B(4,6), X, Y
!hpf$ distribute A(block,*)
!hpf$ distribute B(*,block)
!hpf$ shared B
      ...
```

The array `B` is now shared among all the processors.

The following rules apply for the `SHARED` directive:

- A replicated array cannot be shared. The `SHARED` directive is simply ignored.
- The mapping directive of HPF will be used for the work distribution of parallel loops and array operations.
- Alignments to a shared array will not imply that the aligned array is also shared. A template cannot be shared.

The advantage of a shared array is that the compiler is no longer responsible for emulating the global addresses. It is no longer necessary to compute the owner of one element and to compute a local address on the individual processors.

4.7 The SELECT Directive

The mapping directives of HPF specify the distribution of the arrays, but they do not say anything about the vectorization of operations in the corresponding dimensions.

```
select_directive :  !adp$ select <ident> ( <select_spec_dim_list> )
select_directive :  !adp$ select ( <select_spec_dim_list> ) :: <ident_list>

select_spec_dim  : '*'
select_spec_dim  : selector_list

selector_list    : selector { | selector }*

selector         : NOVECTOR
selector         : VECTOR
selector         : SKIP
selector         : CONCUR
selector         : NOCONCUR
```

The following example shows the use of the SELECT directive for the vectorization.

```
      real, dimension (M,N) :: A, B
!adp$ select A(novector,*) ! do not vectorize operations over rows of A
!adp$ select B(*,vector)   ! vectorize over columns of B
```

In the same way, as operations for the corresponding dimensions will be parallelized if they are distributed, operations for the corresponding dimensions will be vectorized or not. No specification * implies that the compiler can make its own decision.

4.8 Sequence and Storage Association

If an array is distributed the user can make no assumptions about sequence or storage association of this array.

Arrays in common blocks can also be distributed like other arrays. If a common block contains a distributed array, sequence association will not be guaranteed. Such a common block is called nonsequential.

The following rules must apply for a nonsequential common block:

- Every occurrence of the COMMON block has exactly the same number of components with each corresponding component having exactly the identical type, identical shape and the same distribution.
- The definition of the COMMON block must have an occurrence in the main program (used for initialization).

For replicated arrays sequence association is guaranteed. Sequence association can explicitly be specified by the SEQUENCE directive for a COMMON block.

If sequence association is specified, all arrays in the common block will have to be replicated or will be replicated if no layout directive is specified.

```
      COMMON /data/ a(100), b(100,n)
!hpf$ SEQUENCE /data/
```

If no other layout for a and b is given, a and b will become replicated arrays.

5 Default and Underspecified Mappings

A fully specified mapping is given if the mapping is exactly fixed by the mapping directives. This implies that there is not any choice for the compiler about the actual mapping.

An underspecified mapping is given if the mapping directives are not complete. For local arrays with underspecfied mappings the compiler might choose the final mapping that is a specialization of the underspecified mapping (see Section 5.4). For dummy arrays the compiler will generate code that can deal with all mappings of the actuals that are specializations of the underspecified mapping. If the actual mapping is not compatible with the dummy mapping, the data will be redistributed.

5.1 Default Mappings

There are some rules that define a default mapping for arrays. A default mapping in this sense is a fully specified mapping as the compiler will not chose the mapping but apply certain rules.

5.1.1 Defaults for Implicit Mappings

If an array has not an explicit mapping, a default distribution will be assumed for this array. At the moment the user can choose between two possibilities:

- every array is replicated by default,
- or every array is distributed in the last dimensions by default (with up to three distributed dimensions).

In the following situations, an array will not be distributed by default:

- character arrays,
- arrays in derived data types.

5.1.2 Missing ONTO Clauses and Default Processor Arrays

A missing ONTO clause implies a default arrangement. But this arrangement is identical for distributees that have identical shapes and identical explicit mappings.

```
      REAL A(N,N), B(N)
!hpf$ distribute A(block,block)
!hpf$ distribute B(block)
```

By default an abstract processor array is defined for every possible rank. The size of the processor array is always identical to the number of available processors. The shape of the processor array is chosen in such a way that the number of processors is nearly identical for every dimension.

Table 1 shows some examples for initial abstract processor arrays.

If in a distribute directive the `ONTO`-clause is not specified, the distribution will be onto the default processor array whose rank is given by the number of distributed dimensions.

Attention: A missing `ONTO`-clause implies not an underspecified mapping. Only `ONTO *` stands for an underspecified mapping.

NP	Torus 1	Torus 2	Torus 3
5	5	1 x 5	1 x 1 x 5
6	6	2 x 3	1 x 2 x 3
8	8	2 x 4	2 x 2 x 2
12	12	3 x 4	2 x 2 x 3
16	16	4 x 4	2 x 2 x 4
32	32	4 x 8	2 x 4 x 4
33	33	3 x 11	1 x 3 x 11

Table 1: Default abstract processor arrays in ADAPTOR.

5.1.3 Missing Distribution Format

If no distribution formats are specified in a distribute directive, the compiler will chose block distributions for the last n dimensions where n is the rank of the processor array.

```
      program  test ()
!hpf$ processors p(3)
      real a(n,n)
      parameter (n=100)
!hpf$ distribute onto p :: a
```

For this example the compiler will make the following full distribution:

```
!hpf$ distribute (*,block) onto p :: a
```

Attention: It is an error to specify a processor arrangement that has a greater rank than the rank of the distributed object.

5.2 Underspecified Mappings

An underspecified mapping is given in the following situations:

- With the clause `ONTO *` an arbritrary processor arrangement can be specified.

```
      REAL A(N,N)
!hpf$ distribute A (block,block) onto *
```

- The following distribution formats can be used to be unspecific about the distribution of one dimension: `BLOCK()`, `CYCLIC()`, `GEN_BLOCK`, `INDIRECT`, `ALL`
- The following abbreviation can be used:

```
      REAL A(N,N)
!hpf$ PROCESSORS P(4,4)
!hpf$ distribute A * onto P  ! for distribute A(ALL,ALL) onto P
```

An alignment to a target that has an underspecified distribution implies an underspecified mapping for the alignee. But it has to be observed that a specialisation of the distribution of the target will result in a specialisation for the mapping of the alignee.

5.3 Direct Alignments

An alignment of an array to its target is called direct if the following conditions hold:

- dimensions are not permutated (see section 4.4.3),
- there is no embedding or replication (see sections 4.4.5 and 4.4.6),
- the size of an aligned dimension is equal to the size of the corresponding target dimension

Attention: The last condition implies that there is no stride. But there might be an offset.

```
      REAL A(N,N), B(0:N-1,0:N-1,0:N-1)
!hpf$ template T(N,N,N)
!hpf$ align A(I,J) with T(I,J,*)  ! not direct as there is a replication
!hpf$ align A(I,J) with T(2,I,J)  ! not direct as there is an embedding
!hpf$ align B(I,J,K) with T(I+1,J+1,K+1)  ! direct
!hpf$ align B(I,J,K) with T(J+1,I+1,K+1)  ! not direct as permutation
```

A direct alignment implies that the alignment can be replaced by a corresponding distribution if the align_target has a distribution.

5.4 Specialization of Underspecified Mappings

5.4.1 Specialization of Distribution Formats

First we define a notion of specialization for *dist-format*.

1. Each *dist-format* is a specialization of itself. The following equivalences are given:

$$\begin{array}{rcll} \texttt{BLOCK}(n) & \equiv & \texttt{BLOCK}(m) & \text{iff } m \text{ and } n \text{ have the same value} \\ \texttt{CYCLIC}(n) & \equiv & \texttt{CYCLIC}(m) & \text{iff } m \text{ and } n \text{ have the same value} \\ \texttt{CYCLIC} & \equiv & \texttt{CYCLIC}(1) & \end{array}$$

2. `BLOCK` is a specialization of `BLOCK()`, `GEN_BLOCK` and `CYCLIC()`.
3. `BLOCK()` is a specialization of `CYLCIC()`.
4. `CYCLIC(m)` is a specialization of `CYCLIC()`.
5. `GEN_BLOCK(int_array)` is a specialization of `GEN_BLOCK`.
6. `INDIRECT(int_array)` is a specialization of `INDIRECT`.
7. `*` is a specialization of every *dist-format*.
8. Every *dist-format* is a specialization of `ALL`.

5.4.2 Specialization via Distributed Objects

Let `S` and `G` be two named objects that are both distributed by a `DISTRIBUTE` directive.

The distribution of `S` is a *specialization* of the distribution of `G` if the following constraints hold:

1. The shapes of `S` and `G` are the same.
2. The distribution directive of `G` must have a *dist-onto-clause* of "`ONTO` *", or it must have a *dist-onto-clause* specifying the same processor arrangement as that specified in the distribution directive for `S`.
3. Each *dist-format* must be a specialization (in the sense defined above) to the *dist-format* in the corresponding position of the *dist-format-clause* in the distribution directive of `S`.
4. If the onto-clause of the distribution of G specifies a processor arrangement, then '*' can only be a specialization of '*'.

Note: The last condition guarantees that corresponding dimensions of S and G are mapped to the same dimension of the processor array if they are distributed.

```
      REAL S1(N,N), S2(N,N), S3(N,N)
      REAL G1(N,N), G2(N,N), G3(N,N)

!hpf$ distribute S1 (block(10), block) onto P
!hpf$ distribute S2 (block, block) onto Q
!hpf$ distribute S3 (block, CYLCIC) onto Q

!hpf$ distribute G1 (block(),block()) onto P
!hpf$ distribute G2 (block(),block()) onto *
!hpf$ distribute G3 (CYCLIC(),CYCLIC()) onto *
```

For this example, the following relations hold:

- `S1` is a specialization of G1, G2, G3
- `S2` is a specialization of G2, G3 (but not of G1)
- `S3` is a specialization of G3 (but not of G1 and G2)
- `G1` is a specialization of G2
- `G2` is a specialization of G3

```
      REAL S1(N,N), S2(N,N), S3(N,N), S4(N,N)
      REAL G1(N,N), G2(N,N), G3(N,N)

!hpf$ distribute S1 (block, *) onto P
!hpf$ distribute S2 (*, block) onto P
!hpf$ distribute S3 (block, *) onto Q
!hpf$ distribute S4 (*,*)

!hpf$ distribute G1 * onto P
!hpf$ distribute G2 (block(),block()) onto Q
!hpf$ distribute G3 (block(),block()) onto *
```

For this example, the following relations hold:

- S1 and S2 are specializations of G1,
- S3 is a specialiazation of G3, but not of G2
- S1, S2 and S3 are specializations of G3.
- S4 is s specialization of G3

5.4.3 Specialization via Aligned Objects

Let `S` and `G` be two named objects where `G` is distributed by a `DISTRIBUTE` directive and `S` by a `ALIGN` directive.

The mapping of `S` is a *specialization* of the mapping of `G` if all of the following relations hold:

1. The shapes of `S` and `G` are the same.
2. If the distribution directive of `G` has a *dist-onto-clause* that specifies a processor arrangement, then the ultimate target of `S` must also be distributed onto this processor arrangement.
3. Each dimension of `S` is either collapsed or aligned to a certain dimension of its ultimate target. In the last case the dimension of the target must have a *dist-format* that is a specialization of the *dist-format* in the corresponding position of the *dist-format-clause* in the distribution directive of `S`.
4. If the *dist-format* of G is not underspecified, then the corresponding dimension of S must be directly aligned (same size as the align target).
5. If the onto-clause of the distribution of G specifies a processor arrangement, then '*' can only be a specialization of '*'.
6. If the onto-clause of the distribution of G specifies a processor arrangement, then the corresponding dimensions of G and S must map to the same dimension of the processor arrangement.

The above definition is very general. It says that the mapping of an aligned array can also be a specialization if it contains permutations, linear embeddings, embedded dimensions or replicated dimensions. It must only be guaranteed that the dimension to which the dimension of the alignee is aligned to must be distributed in a certain way.

```
!hpf$ template, dimension (N2,N2) :: T
!hpf$ distribute (block, block) onto P
      real, dimension (N,N) :: A1, A2, A3
!hpf$ ALIGN A1(I,J) with T(I,J)
!hpf$ ALIGN A2(J,I) with T(J,I)
!hpf$ ALIGN A3(I,J) with T(2*I,2*J)

      real, dimension (N,N) :: G
!hpf$ distribute G (block(),block()) onto P
```

- `A1`, `A2` and `A3` are all specializations of `G`.

5.4.4 Aligmments to Underspecified Distributions

If an array is aligned to a target that has an underspecified distribution, its mapping is in a certain sense also underspecified. Nevertheless, it makes no sense to define a specialization of the aligned array without considering the target. As soon as the distribution of the target has been specialized, it implies a specialization of the alignee.

```
      REAL G1(N,N), G2(N,N)

!hpf$ distribute G1(block(),block()) onto *
!hpf$ align G2(I,J) with G1(I,J)
```

In this example, the object `G2` is aligned with the object `G1`. As the distribution of `G1` is underspecified, the distribution of `G2` is also underspecified. In case of local arrays, the compiler will choose a fully specified mapping for G1 and G2. As soon as the mapping of G1 is fixed, it implies a mapping of G2.

```
      real, dimension (N,N) :: S1, S2, S3
      real, dimension (N,N) :: G1, G2

!hpf$ distribute S1(block, block) onto P
!hpf$ align S2(I,J) with S1(I,J)
!hpf$ distribute S3(block,block) onto Q

!hpf$ distribute G1(block(),block()) onto *
!hpf$ align G2(I,J) with G1(I,J)
```

Attention: The HPF 2.0 standard says that S3 is a specialiazation of G1 and S1 is also a specialization of G2. But S1 is obviously not aligned to S3.

6 Data Mapping in Subprogram Interfaces

6.1 Introduction

The data mapping features described in section 4 can also be used to describe the mapping of dummy arguments.

The mapping of each such dummy argument may be related to the mapping of its associated actual argument in the calling main program or procedure (the "caller") in several different ways.

full specified mapping The directive describes the mapping of the dummy argument. However, the actual argument need not have this mapping. *If it does not*, it is the responsibility of the compiler to generate code to remap the argument as specified, and to restore the original mapping on exit. This code may be generated in either the caller or in the called subprogram.

underspecified mapping The mapping is underspecified. The called subprogram must accept the mapping of the actual argument if it is a specialization of the mapping of the dummy argument (see section 5.4). Otherwise, remapping takes place in the same way as for fully specified mappings.

inherited mapping The dummy argument has the `INHERIT` attribute. The called subprogram must accept every mapping of the actual argument, a redistribution takes never place.

With the `RANGE` directive the user can be more specific about the possible mappings.

6.2 What Remapping is Required, and Who Does It

The ADAPTOR compiler will generate the code for the remapping in the called subprogram. So it is usually not necessary to provide an explicit interface for the called subprogram in the caller. This approach has been chosen for the following reasons:

- If the subprogram needs shadow edges, an additional local copy of data can be avoided.
- If the subprogram cannot deal with every actual mapping that is a specialization of the dummy argument, it might chose a more specific distribution. As the routine itself is responsible for the remapping, double remappings can be avoided.

Therefore it is not absolutely necessary to provide an explicit interface. Nevertheless, it is recommended for the following reasons:

- HPF 2.0 requires explicit interfaces if arguments are remapped. Therefore explicit interfaces increase the portability.

- An explicit interface will allow that the caller can also do the remapping. For optimizations, this might sometimes be more useful.

Attention: Within local and serial routines (`EXTRINSIC("LOCAL",...)`,`EXTRINSIC("SERIAL",...)`), the called routine will not do any remapping. In this case, an explicit interface is mandatory if a redistribution is necessary.

6.3 Inherited Mappings and the Range Directive

If the `INHERIT` attribute is specified for a dummy argument, the called subprogram must accept any mapping of the actual argument. The actual argument will not be redistributed. Therefore a dummy argument with the `INHERIT` attribute must not appear as an alignee in an `ALIGN` and not as a distributee in a `DISTRIBUTE` directive.

The `RANGE` directive is used to restrict the possible mapping formats of the actual argument.

```
      SUBROUTINE SUB (X)
      REAL X(:,:)
!hpf$ INHERIT X
!HFP$ RANGE X (block(),block()) (*,GEN_block())
```

The object in the `RANGE` directive must have the `INHERIT` attribute. The mapping of the actual argument must be a specialization of at least one of the *format-clauses* in the `RANGE` directive.

Attention: There is no runtime check to verify that the actual argument has really a certain mapping

```
         REAL A(100, 100, 100)
!hpf$    distribute A(block, *, CYCLIC)

         CALL SUB( A(:,,:,1) )          ! Conforming
         CALL SUB( A(:,,1,:) )          ! Nonconforming
         CALL SUB( A(1,,:,:) )          ! Nonconforming
            ....

         SUBROUTINE SUB(X)
         REAL A(:, :)
!hpf$    INHERIT X
!hpf$    RANGE X (block, *)
```

6.4 Passing Array Sections

In the most situations, ADAPTOR will use copy-in and copy-out for array sections passed to subprograms. The only exception is given for inherited mappings.

```
      REAL a(n,n)
      ...
      CALL sub (a(2:n:2,:),a(1:n:2,:))
      ...

      SUBROUTINE sub (x, y)
      REAL x(:,:), y(:,:)
!hpf$ distribute x(block(),block()) onto *
!hpf$ INHERIT y
      ...
      END SUBROUTINE sub
```

6.5 Some Remarks about Efficiency

The more underspecified a distribution is, the more general is the code.

- For fully specified mappings the compiler can verify that two objects have the same mapping.

```
      SUBROUTINE sub (A1, A2, N)
      REAL A1(N), A2(N)
!hpf$ distribute (block) :: A1, A2
      A1 = A2    ! same distribution, no communication
      END SUBROUTINE sub
```

- For underspecified mappings (here the missing *onto-clause*) the compiler must assume that A1 and A2 are distributed onto different processor arrangements.

```
      SUBROUTINE sub (A1, A2, N)
      REAL A1(N), A2(N)
!hpf$ distribute (block) onto * :: A1, A2
      A1 = A2    ! distribution can be different (communication)
      END SUBROUTINE sub
```

- For underspecified mappings the `ALIGN` directive should be used to indicate relation between data.

```
      SUBROUTINE sub (A1, A2, N)
      REAL A1(N), A2(N)
!hpf$ distribute (block) onto *:: A1
!hpf$ align (I) with A1(I) :: A
      A1 = A2    ! aligned, no communication
      END SUBROUTINE sub
```

The following example shows how the mechanism can be used to realize efficient code for different distributions.

```
      integer, parameter :: N1=10, N2=2*N1, N3=4*N1, N4=8*N1

      real, dimension (0:N1,0:N1) :: A1
      real, dimension (0:N2,0:N3) :: A2
      real, dimension (0:N3,0:N3) :: A3
      real, dimension (0:N4,0:N4) :: A4

!hpf$ align A1(I,J) with A4(8*I,8*J)
!hpf$ align A2(I,J) with A4(4*I,4*J)
!hpf$ align A3(I,J) with A4(2*I,2*J)

      CALL sub (a1,a2,N1)
      CALL sub (a2,a3,N2)
      CALL sub (a3,a4,N3)

      SUBROUTINE sub (x, y, n)
      real x(0:n,0:n), y(0:2*n,0:2*n)
!hpf$ distribute y(block(),block()) onto *
!hpf$ align x(i,j) with y(2*i,2*j)
      ...
      END SUBROUTINE sub
```

- The actual mappings are always specializations of the dummy mapping of `x`, the alignment of the actual argument for `y` is also true. So there will never be redistributions.
- The code for the subroutine will be still very efficient.
- This example might not be efficient when using inherited distributions with the `RANGE` directive.

6.6 Differences to HPF

Due to the fact that this kind of directives could be combined and the new standard allows also underspecified mappings, we gave up this terminology and introduced a terminology based on underspecified mappings.

- A full specified mapping for a dummy argument is like the *prescriptive* directive of HPF.
- A *descriptive* directive describes the mapping of the dummy. It is the responsibility of the caller to insure that the actual as passed has this mapping. Similarly, remapping to restore the original mapping on exit must also be done by the caller. HPF 2.0 does not make any longer a difference between descriptive and prescriptive directives, an explicit interface is always required.
- A *transcriptive* directive is used to let the mapping unspecified. The called subprogram must accept the mapping of the argument as it is passed. Of course this means that the caller must pass this mapping information at run-time. This corresponds to the underspecified mappings of dummy arguments.

7 Data Parallelism

7.1 Overview of Data Parallelism

Fortran 90 and Fortran 95 contain already a lot of features to specifiy explicit data parallelism:

- Array syntax and array statements,
- the `FORALL` statement and `FORALL` construct,
- intrinsic functions on arrays.

This data parallelism is already used by ADAPTOR to generate a parallel program.

HPF provides the following addtional possibilities for specifying explicit data parallelism:

- Array syntax and array statements,
- the `FORALL` statement and `FORALL` construct,
- the `INDEPENDENT` directive,
- extended intrinsic functions and standard library.

Figure 11 shows the semantic of the `FORALL` statement and the `INDEPENDENT` loop compared to the serial loop. The execution order is less restrictive and increases the potential of parallel execution.

Attention: ADAPTOR has no features for automatic parallelization of serial loops.

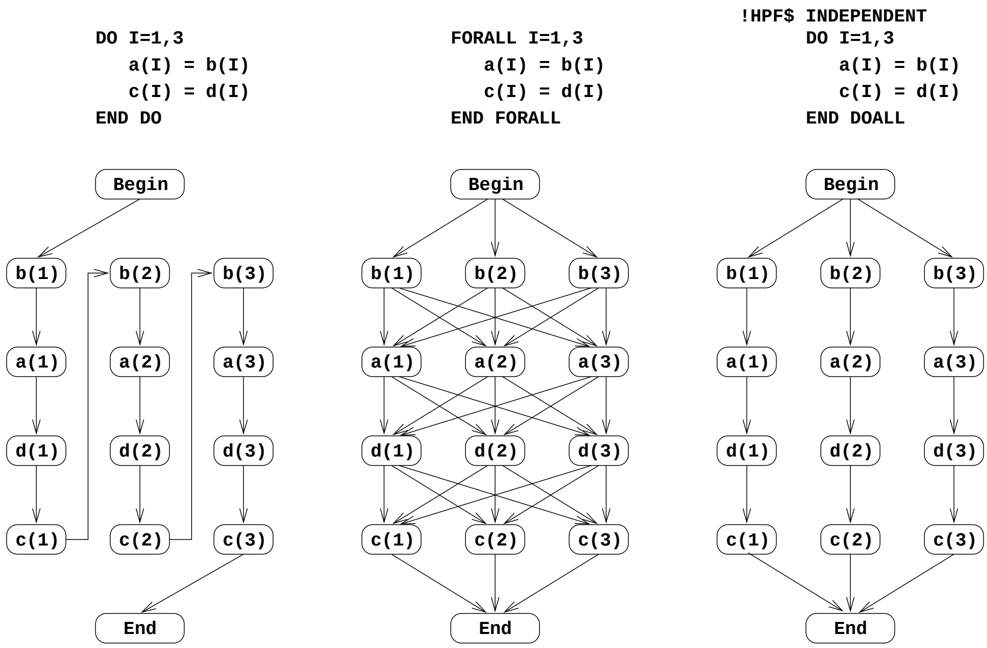


Figure 11: Execution order for serial and parallel loops.

7.2 The INDEPENDENT Directive

The `INDEPENDENT` directive can precede an indexed `DO` loop or `FORALL` statement or construct. It asserts to the compiler that the operations in the following `FORALL` statement or construct or iterations in the following loop may be executed independently – that is, in any order, or interleaved, or concurrently – without changing the semantics of the program.

```
!hpf$ INDEPENDENT
   DO i = 1, n
      x(i) = a(i) * a(i)
      d(i) = x(i) + c(i)
   END DO
```

The `INDEPENDENT` directive can also be used for the `FORALL` statement.

```
!hpf$ INDEPENDENT
      FORALL (i=1:n) a(ind1(i)) = a(ind2(i))
```

In the context of a `DO` loop, the `INDEPENDENT` directive can be combined with some other options:

- the `NEW` option specifies variables that become private for one iteration (see also section 12.8),
- the `REDUCTION` option specifies variables whose results will be combined at the end of the parallel loop (see section 13.4),
- the `ON HOME` option can be used to specify the home of the iteration, where the iteration is executed,
- the `RESIDENT` assertion guarantees that no non-local data is referenced within one iteration.

These additional directives are only allowed for `DO` loops, but not for the `FORALL` statement or construct.

7.3 The ON Directive

The `ON` direcitve allows the programmer to control the distribution of computations among the processors of a parallel machine. The `ON` directive specifies an active processor set for a statement or a set of statements.

```
!hpf$ on (variable)          !hpf$ on (processors-elmt)
      <stmt>                       <stmt>


!hpf$ on home (variable) begin          !hpf$ on (processors-elmt) begin
      <stmt>                                  <stmt>
!hpf$ end on                            !hpf$ end on
```

The `ON` directive restricts the active processor set for a computation to those processors named in its home. The home can also be a single processor.

The home can name a variable, a template, or a processors arrangement. For each of these possibilities, it can specify a single element or multiple elements. This is translated into the processor(s) executing the `ON` block as follows:

- If the home clause names a program object, then every processor owning any part of that object should execute the `ON` block.
- If the home clause names a processor arrangement, then the processor(s) referenced there should execute the `ON` block. E.g., for `ON P(2:4)` the statement will be executed on the three processors `P(2)`, `P(3)`, and `P(4)`.

Attention: ADAPTOR makes only a difference between a single element in the dimension and the full dimension. `ON HOME(A(2:4))` is therefore the same as `ON HOME(A(:))`, where `ON HOME (A(2))` exactly specifies the processors owning the second element.

```
        real, dimension (N) :: A1, A2
!hpf$   processors PROCS(4)
!hpf$   distribute A1 (block) onto PROCS(1:2)
!hpf$   distribute A2 (block) onto PROCS(3:4)
        ...
!hpf$   on (PROCS(1:2))
        call TASK1 (A1,N)
!hpf$   on (PROCS(3:4))
        call TASK2 (A2,N)
```

7.4 The REDUCTION Directive

In contrary to the HPF 2.0 language definition, ADAPTOR allows the completely independent use of the *REDUCTION* directive.

```
!hpf$ REDUCTION (reduction_vars)
      <statement>


!hpf$ REDUCTION (reduction_vars) BEGIN
      <block>
!hpf$ END REDUCTION
```

8 Task Parallelism

8.1 The TASK_REGION Construct

The TASK_REGION construct allows the user to specifiy that disjoint processor subsets can execute blocks of code concurrently.

```
       real, dimension (N,N) :: A1, A2

!hpf$   processors PROCS(4)
!hpf$   distribute A1 (*,block) onto PROCS(1:2)
!hpf$   distribute A2 (*,block) onto PROCS(3:4)

       ! define a task region, otherwise home will be ignored

!hpf$   task_region
!hpf$     on home (A1), resident
            call TASK1 (A1,N)
!hpf$     on home (A2), resident
            call TASK2 (A2,N)
!hpf$   end task_region
```

The task region has some advantages:

- clear specification where task parallelism appears,
- it provides syntactical restrictions (every `ON` directive must be combined with the `RESIDENT` directive),
- the user guarantees no I/O interferences between the different execution tasks.

The data parallel tasks *TASK1* and *TASK2* will be executed independently on different processor subsets.

Actually, ADAPTOR provides no mechanism to map scalar data to a processor subset. Therefore it should be avoided to pass scalar data to a task that will be modified.

8.2 Data Parallel Pipelines

If there are data dependences between the tasks of a task region, these dependences are observed and will lead to a serial execution. Nevertheless, it can be used to realize a pipeline between data parallel tasks if the task region is executed within a loop.

```
!hpf$   processors PROCS(P)
       real, dimension(N,N) :: A1,A2
!hpf$   distribute A1 (*,block) onto PROCS (1:P1)
!hpf$   distribute A2 (block,*) onto PROCS (P1+1:P)
       ...
       do I = 1, 10
!hpf$   task_region
!hpf$     on home (A1), resident
            call TASK1 (A1, N)
          A2 = A1
!hpf$     on home (A2), resident
            call TASK2 (A2, N)
!hpf$   end task_region
       end do
```

8.3 The HPF_TASK_LIBRARY

Communication between different data parallel tasks might be possible if there are no dependences between the different tasks and the tasks are executed on disjoint processor subsets. The tasks automatically get a task identifier starting with 1:

```
!hpf$ independent task_region
!hpf$ on (PROCS(1))                     ! will be task 1
        call STAGE1 ()
!hpf$ on (PROCS(2:3))                   ! will be task 2
        call STAGE2 (2)
!hpf$ on (PROCS(4:5))                   ! will be task 3
        call STAGE2 (3)
!hpf$ on (PROCS(6))                     ! will be task 4
        call STAGE3 ()
!hpf$ end task_region
```

Within a data parallel task, the `HPF_TASK_LIBRARY` can be used to communicate between different tasks.

```
subroutine STAGE1 ()
use HPF_TASK_LIBRARY
...
end subroutine STAGE1
```

The following subroutines support the initialization and termination of data parallel tasks. They might already be called implicitly when the data parallel tasks are invoked.

```
subroutine HPF_TASK_INIT ()

subroutine HPF_TASK_EXIT ()
```

The call of these routines is not mandatory but might assert additional runtime checks. They could verify at runtime that the tasks of the current context are really mapped to disjoint processor subgroups. Furthermore, at the end it could be verified that there are no pending messages between the tasks.

The following subroutines return the size (number of data parallel tasks in the current context) and the rank of the calling task ($1 \leq rank \leq size$).

```
subroutine HPF_TASK_SIZE (size)
integer, intent(out) :: size

subroutine HPF_TASK_RANK (rank)
integer, intent(out) :: rank
```

For the sending of data (scalars, arrays or array sections), the task identifier of the target task must be specified.

```
subroutine HPF_SEND (data, dest)
integer, intent (in) :: dest
<type>,  intent(in)  :: data
```

The receiving of data is similiar. Every send must have a matching receive.

```
subroutine HPF_RECV (data, source)
integer, intent (in), optional :: source
<type>,  intent(out)  :: data
```

The source argument is optional. By this way, it is possible to receive a message from an arbitrary task.

The implementation of point-to-point communication between data parallel tasks results in communication between the processors of the two processor subgroups that are involved. If distributed data is exchanged, it is necessary to exchange the mapping information (*descriptor exchange*).

The following restrictions are given:

- If the tasks are not really executed concurrently, the code might result in a deadlock.
- Every send must have a corresponding receive as otherwise the communication will conflict with compiler generated communication outside the task region.
- Only scalar data can be received from any processor.

The following routines are helpful to avoid the descriptor exchange when the same schedule is used several times.

```
subroutine HPF_SEND_INIT (data, dest, request)
integer, intent (in)  :: dest
<type>,  intent (in)  :: data

subroutine HPF_RECV_INIT (data, source, request)
integer, intent (in)  :: source
<type>,  intent (out) :: data
integer, intent (out) :: request

subroutine HPF_TASK_COMM (request)
integer, intent (in)  :: request
```

9 Extrinsic Procedures

ADAPTOR supports the EXTRINSIC facilities of HPF for serial and local routines. Local and serial routines can be written in FORTRAN 77 or in HPF.

local routines:
: a local routine allows to write single-processor code that works only on data that is mapped to a given physical processor. In this sense, a local routine contains only local computations (see section 12).

  Nevertheless it might be possible to call message passing commands within a local routine, but in this case the user himself is responsible for any communication. But only local data can be accessed within the local routine. Local routines can also be used to provide an HPF interface to implementation-specific parallel libraries.

serial routines:
: a serial routine allows to write code that will only be executed by a single physical processor.

9.1 HPF_LOCAL Procedures

HPF_LOCAL routines allow to write code that only works on local part of the arrays. ADAPTOR does not act on mapping directives of the arrays within the routine, so the code will work for all kind of actual arguments.

```
extrinsic (HPF_LOCAL) subroutine S (A, COL)
real, dimension :: A(:,:)
integer :: COL
....                 ! code that works on the local part of A
end subroutine S
```

From the caller's standpoint, an invocation of a local procedure from a "global" HPF program has the same semantics as an invocation of a regular procedure.

If one processor is executing a local routine, it might be possible that only some processors are executing this incarnation. Therefore in a local routine it is not possible to have any global operation like redistribution or calling global routines. So we have the following restrictions for local routines:

- Within a local routine only other local routines can be called. It is not possible to call any global routine.
- Every mapping directive is considered as a descriptive directive. There can be never any redistribution within a local routine.

As already mentioned, the user may:w all any message passing code within local routines. But then he himself is responsible that the routine is executed by all participating processors.

9.1.1 HPF Local Routine Library

Local HPF procedures can use any HPF intrinsic or library procedures. In addition, a set of local library routines are provided that can be used to get more information about the global view of the actual arguments.

```
use HPF_LOCAL_LIBRARY
```

- The routines `GLOBAL_ALIGNMENT`, `GLOBAL_DISTRIBUTION`, and `GLOBAL_TEMPLATE` can be used to query the global mapping of an actual argument while the corresponding HPF routines would return the local mapping that is the trivial one.
- The local library function `MY_PROCESSOR` returns the identifier P of the calling processor with $0 \leq P < NP$ where NP is the number of processors returned by the global `HPF_LIBRARY` function `NUMBER_OF_PROCESSORS`.
- `ABSTRACT_TO_PHYSICAL`
- `PHYSICAL_TO_ABSTRACT`
- `LOCAL_TO_GLOBAL`
- `GLOBAL_TO_LOCAL`
- `LOCAL_BLKCNT`
- `LOCAL_LINDEX`
- `LOCAL_UINDEX`
- `GLOBAL_SIZE` returns the global size of an actual argument.

9.1.2 Message Passing in HPF_LOCAL Routines

Processors executing a local HPF routine can communicate with each other. The context is given by the last global HPF context in which the local routine has been called.

The routines are also provided with the HPF task library (see section 8.3). So the routines in the local model have the same syntax as the corresponding routines for communication between data paralllel tasks.

- The subroutine TASK_SIZE returns the number of processors executing the same local subroutine. This number corresponds to the value of the global HPF_LIBRARY function ACTIVE_NUM_PROCS as if it has been called before the call of the local routine (within the local routine the number of active processors is 1). The subroutine TASK_RANK returns the corresponding id $1 \leq id \leq P$ where P is the value return by TASK_SIZE.
- Data can be sent by the routine HPF_SEND, while HPF_RECV allows the receiving of data. In contrary to the corresponding routines of the HPF_TASK_LIBRARY, no exchange of array descriptors is required as only communication between single processors is executed.

9.2 HPF_SERIAL Procedures

9.2.1 Example for Serial Extrinsics

In some situations it is necessary that only one node calls a certain subroutine, e.g. for X-Windows routines.

```
interface

   extrinsic (hpf_serial) subroutine x_display_init (width, height)
   integer width, height
   intent (in) :: width, height
   end

   extrinsic (hpf_serial) subroutine x_show_image (image, width, height)
   integer image (height, width)
   integer height, width
   intent (in) :: width, height, image
   end subroutine

   extrinsic (hpf_serial) subroutine x_new_action (hx1, hx2, hy1, hy2)
   integer hx1, hx2, hy1, hy2
   intent (inout) :: hx1, hx2, hy1, hy2     ! default is in-out
   end

   extrinsic (hpf_serial) subroutine x_display_exit
   end

end interface
```

9.2.2 Invoking a Serial Routine

A serial routine will always be called by a single processor only. The compiler generates communication to make sure that all parameter data is availabe on this processor. A serial procedure can also be called with distributed data. In this case, the calling routine will generate an incarnation of the full array on the node that will call the subroutine. After the call of the serial procedure all replicated data will be broadcast and the distributed data will be restored. Information about the intent of the arguments will be used to decide about the necessity.

Usually, a serial procedure will be executed by the first processor that executes also all I/O routines (in this sense I/O operations are very similiar to serial routines). But if all are arguments are distributed on one processor, that processor will run the call.

```
      real A(N), B(N)
!hpf$ distribute (block) :: A, B
      ...
      call SERIAL (A(I), B(I))   ! will be executed by owner of A(I), B(I)
```

In this example, the routine `SERIAL` runs on the processor that owns `A(I)` and `B(I)`. If such a call is within a loop and at each iteration a different processor is used, efficient parallelism is given.

9.2.3 Interface for HPF_SERIAL Routines

Attention: An interface block is usually mandatory to guarantee in the calling routine that the routine is executed on a single processor. Only if the user specifies a home for the subroutine call that corresponds to a single processor and if the compiler accepts it, it is not necessary.

In a certain sense, a serial routine corresponds to a data parallel routine that is executed on a single processor. The calling routine has to guarantee the locality of the arguments.

9.2.4 Execution of HPF_SERIAL Routines

As a serial routine is always executed on a single processor, the HPF compiler has not to generate any communication. All actual arguments are mapped to or have been mapped to the processor executing the routine. Local variables become private variables.

9.2.5 Access to Global Data in Serial Routines

The HPF standard does not allow to share global data (e.g. `COMMON` blocks) between serial and global HPF routines, for many good reasons. If global data is updated in a serial routine on a single processor, this might result in inconsistencies. On the other hand, it is very useful to read replicated global data in the serial routine. This is supported within ADAPTOR.

```
      real A(N), B(N)        ! replicated data
      common /DATA/ A, B
      read *, A, B
      ...
      call SERIAL ()   ! will be executed by one processor
      ...

      extrinsic (HPF_SERIAL) subroutine SERIAL ()
      real A(N), B(N), X
      common /DATA/ A, B
      X = A(5)          ! might be useful
      A(3) = X          ! not HPF conform as only one processor updates
      ...
      end subroutine SERIAL
```

Furthermore, ADAPTOR will also generate code when global distributed data is accessed. Again, the user is responsible that the serial routine only accesses data that is local on the executing processor. This corresponds to the access of global data in a data parallel routine that is only executed on a subset of processors.

```
       module DATA
       integer, paramter :: N = 100
       real, dimension (N, N) :: A
!hpf$  distribute A(*,block)
       end module DATA

       program WORK
       use DATA
       ...
```

```
!hpf$  independent, on home (A(:,I))
       do I = 1, N
          call WORK_COL (I)
       end do
       ...
       end program WORK

       extrinsic (HPF_SERIAL) subroutine WORK_COL (I)
       use DATA
       integer I
       do K = 1, N
          A(I,g(K)) = f(A(I,h(K))
       end do
       end subroutine WORL_COL
```

As this example shows, the serial routine is used to take advantage of the fact that all data is resident when the routine is executed on the processor owning the i-th column of the distributed array A.

9.2.6 Library Access from Serial Extrinsics

Within a serial HPF program, all HPF_LIBRARY routines can be called. The HPF_LOCAL_LIBRARY module must not be used. Message passing within a serial routine is not useful as the routine is executed within a context that consists of a single processor.

9.3 F77_LOCAL Procedures

The main difference between a HPF_LOCAL and a F77_LOCAL routine is how arrays are passed. While in the HPF model usually both, pointers and HPF array descriptors ("handles"), are passed, the F77 routine will only expect one of them. This is simply due to the fact that HPF local routines will be compiled by the HPF compiler (knowing about descriptors) while F77 local routines are compiled by a FORTRAN 77 compiler.

- LAYOUT('F77_ARRAY') (the default) vs. LAYOUT('HPF_ARRAY')
- PASS_BY('*') (the default) vs. PASS_BY('HPF_HANDLE')

The FORTRAN 77 Local Library is available. This includes the HPF-callable subgrid inquiry subroutine HPF_SUBGRID_INFO as well as the FORTRAN 77-callable inquiry subroutines:

- F77_SUBGRID_INFO,
- F77_GLOBAL_ALIGNMENT,
- F77_GLOBAL_DISTRIBUTION,
- F77_GLOBAL_TEMPLATE,
- F77_ABSTRACT_TO_PHYSICAL,
- F77_PHYSICAL_TO_ABSTRACT,
- F77_LOCAL_TO_GLOBAL,
- F77_GLOBAL_TO_LOCAL,
- F77_LOCAL_BLKCNT,
- F77_LOCAL_LINDEX,

- F77_LOCAL_UINDEX,
- F77_GLOBAL_SHAPE,
- F77_GLOBAL_SIZE,
- F77_SHAPE,
- F77_SIZE, and
- F77_MY_PROCESSOR.

If an HPF routine calls a local F77 routine, an interface must be available (exceptions for ADAPTOR are explained later).

If all of the following conditions are true, an interface is not necessary (at least for ADAPTOR):

- There is no redistribution at the subprogram boundary.
- The F77 routine is a subroutine and not a function.
- The F77 routine does not expect any character argument.

9.4 F77_SERIAL Procedures

A F77 serial routine will be called only by one processor. The arguments itself are passed in the same way as for F77 local routines.

The inquiry subroutines for F77 local routines can also be called within serial F77 routines.

10 HPF Intrinsic and Library Procedures

10.1 HPF Intrinsic Procedures

The system inquiry functions `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE` are supported by ADAPTOR.

The system inquiry functions `ACTIVE_NUM_PROCS` and `ACTIVE_PROCS_SHAPE` of the approved extensions are supported by ADAPTOR. The following rules apply:

- `ACTIVE_NUM_PROCS` returns the total number of processors executing the program the number of processors exectuing the program along a specified diemsnions of the processor array as determined by the innermost `ON` block.
- `ACTIVE_NUM_PROCS` returns always 1 in serial or local routines.

The computational intrinsic function `ILEN` is not supported.

The extended version of `TRANSPOSE` of the HPF approved extensions is not supported.

ADAPTOR supports currently all subroutines and computational functions of the HPF library. It is absolutely necessary to have an appropriate `USE` statement, otherwise the routines are not handled correctly.

The mapping inquiry subroutines `HPF_ALIGNMENT`, `HPF_TEMPLATE`, and `HPF_DISTRIBUTION` are now available in ADAPTOR as this tool now supports also inherited distributions.

The bit manipulation functions LEADZ, POPCNT and POPPAR are not supported.

The new reduction functions IALL, IANY, IPARITY and PARITY can be used with ADAPTOR.

With ADAPTOR, the array combining scatter functions XXX_SCATTER are full supported and implemented efficiently.

Array prefix and suffix functions XXX_PREFIX and XXX_SUFFIX are not supported by ADAPTOR.

The array sort functions GRADE_DOWN, GRADE_UP, SORT_DOWN and SORT_UP are not supported.

11 Execution Model of HPF Programs

This section describes the execution model of High Performance Fortran. Though the description is related to the ADAPTOR compilation system, most of the techniques might also be applied within other HPF compilers.

11.1 Serial Execution of HPF Programs

HPF programs can also run as serial programs. For this purpose, ADAPTOR treats them as follows:

- Directives are completely ignored, every array has exactly one full incarnation and independent loops are executed serially.
- The FORALL statement and construct are serialized (this might cause the introduction of temporary data).

  ```
  FORALL (I=2:N-1) A(I) = f(A(I-1),A(I+1))
  ```

  is translated to

  ```
  ALLOCATE (TMP(2:N-1))
  DO I = 2, N-1
     TMP(I) = f(A(I-1),A(I+1))
  END DO
  DO I = 2, N-1
     A(I) = TMP(I)
  END DO
  DEALLOCATE (TMP)
  ```

- Calls for INTRINSIC or HPF Library routines are replaced with calls to a special library version that is available for serial execution.

11.2 The SPMD Model

The essential idea of the source-to-source-translation is to assign the arrays of the source program to the memory of the node processors as specified by the HPF mapping directives (see Figure 12).

The control flow and statements with scalar code are replicated on all nodes. The parallel loops and array operations are restricted to the local part owned by the processor. Communication statements for exchanging non local data will be generated automatically as subroutine calls to the runtime system.

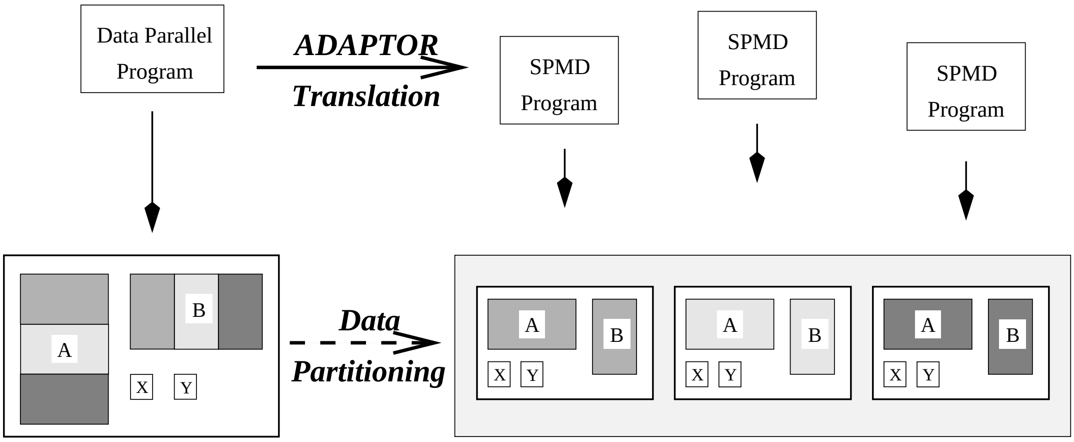


Figure 12: SPMD execution of HPF programs.

11.3 Home of Computations

Beside the `ON HOME` directive considered for the next HPF standard, HPF allows only to specify the distribution of data but not to specify the work distribution. For this reason, the compiler has to decide which statements will be executed by which processor.

The main criteria for the load distribution are:

- An assignment to a distributed variable will be executed by the processor that owns the element on the left hand side (owner-computes rule). The needed data (values on the right hand side) must be made available before.
- An assignment to a scalar or replicated variable will be executed by every processor.
- I/O statements are executed by one dedicated processor (see section 11.6).
- One iteration of an independent loop is executed completely on the home of this iteration. The ownership is usually given by the first assignment to a distributed variable.

```
!hpf$ INDEPENDENT
      DO i = 1, n
         s    = ....
         x(i) = f(s, ...)
         ....
      END DO
```

  Iteration `i` is executed by the processor that owns the element `x(i)`.

For optimization issues it might be the case that there are exceptions from these rules.

11.4 Active Processors

An active processor is one that executes an HPF statement or block. Usually, an HPF program begins execution with all processors active. But the execution can be restricted to a subset of processors or to a single processor in the follwoing cases:

- The `ON` directive (see section 7.3) restricts the active processor set for the duration of execution of statements in its scope.
- Serial routines (see section 9.2) will be executed on a single processor.

11.5 Execution of Subroutines

Usually, every user subroutine and every user function will be entered by all processors. These are the exceptions:

- A serial routine is only executed by a single processor.
- A pure subprogram is only executed by the processor that will call it. If it is called in a parallel loop, only one processor will execute it.
- If the home is specified, the call is assumed to be pure and the previous rules apply.

From the caller's standpoint, an invocation of a local procedure from a "global" HPF program has the same semantics as an invocation of a regular procedure.

11.6 I/O

ADAPTOR does not support parallel I/O until now. Currently I/O statements are translated in such a way that I/O operations are executed by the first node process. In the following, this process is called I/O process.

Due to the fact that only one process executes I/O statements there are some inconveniences that should be observed.

- There are no problems when using replicated data in I/O statements. If a replicated variable is updated by the statement (e.g. `READ` or in `INQUIRE`), the value is broadcast to all other processors (implicit communication and synchronization).
- If distributed arrays are used in an I/O statement, the corresponding values have to be send to or to be received from the I/O process (implicit communication). Therefore temporary data of the corresponding size will be created.

```
      REAL A(N,N)
!hpf$ distribute A(block,block)
      ...
      READ *, A
```

  is translated to

```
      ALLOCATE (TMP_A (N,N))
!hpf$ distribute TMP_A (*,*)
      READ *, TMP_A
      A = TMP_A
      DEALLOCATE (TMP_A)
```

Attention: As the full array is allocated on a single node, there might be memory problems with large arrays. In such a case, it is recommended to read single columns or rows of a matrix.

12 Local Computation

The central idea of generating efficient HPF programs is to ensure data locality. The more operations that are performed between operands which reside on the same processor, the more efficient the implementation will be.

Therefore this section will show which data parallel statements need no communication and will be most efficient for parallel execution.

Attention: It is very important that already at compile time it can be recognized which operands are on the same processor. Otherwise the compiler has to generate additional queries about data locality which can be very expensive.

12.1 Local Array Assignments

```
      REAL a(n,n), a1(n,n)
!hpf$ distribute a(*,block) :: a, a1
      REAL g

      a = a1                  ! no communication, same distribution

      a = g                   ! g is replicated on all processors

      a(1,1:n) = a(2,1:n)     ! no communication
```

12.2 Local FORALL Statements

Similar to the array statements, `FORALL` statements will be identified as local statements at compile time if data is aligned or mapped to the same processor.

```
      REAL a(n,n), b(n,n)
!hpf$ distribute (*,block) :: a, b
      ...
      FORALL (i=1:n, j=1:n) a(i,j) = 1.0 / REAL(i+j-1)
      FORALL (i=1:n, j=1:n, a(i,j) .NE. 0.0) b(i,j) = 1.0 / a(i,j)
```

12.3 Local Independent Loops

With a local independent `DO` loop it is possible to specify that all iterations of the loop can be executed independently and no communication is necessary. This kind of loop was mainly intended for internal representations within ADAPTOR as many other kind of array statements and parallel loops will be translated internally to such loops. In some situations however, it might be useful to use this kind of loop at user level.

Attention: The `RESIDENT` directive can only be used with the `ON_HOME` directive.

```
!hpf$ INDEPENDENT, RESIDENT, ON HOME x(i)
   DO i = 1, n
      x(i) = a(i) * a(i)
      d(i) = x(i) + c(i)
   END DO
```

12.4 Using Alignment

By the `ALIGNMENT` directive arrays can be mapped in such a way that a computation does not need any communication.

```
      REAL a(0:2*n), b(0:n)
      REAL a1 (1:2*n-1)
!hpf$ align b(i) with a(2*I)
!hpf$ align a1(i) with a(i)
      ...
      FORALL (i=0:n) b(i) = a(2*i)
      FORALL (i=1:2*n-1) a1(i) = a()
```

In this example the compiler will know at compile time that no communication is necessary. This would not be the case without the `ALIGNMENT` directives.

If two arrays are distributed in the same way, a local computation can be identified if the size of the arrays is known at compile time. For allocatable arrays, the size might not be known at compile time.

```
      REAL, ALLOCATABLE :: A(:,:), B(:,:)
!hpf$ distribute (block,block) :: A, B
      ...
      FORALL (I=1:N,J=1:N) A(I,J) = B(I,J)
```

This `FORALL` statement cannot be identified as local computation because ADAPTOR must assume that arrays A and B have different sizes. With the `ALIGN` directive, the computation can be specified as a local one.

```
      REAL, ALLOCATABLE :: A(:,:), B(:,:)
!hpf$ align B(I,J) with A(I,J)
      ...
      FORALL (I=1:N,J=1:N) A(I,J) = B(I,J)
```

12.5 Using Shared Arrays

For `INDEPENDENT` loops the HPF compiler must usually insert communication for all data that is non-local to the processor that executes the corresponding iteration of the loop.

```
!hpf$ INDEPENDENT
      DO I = 1, N
         A(f(I)) = B(g(I))
      END DO
```

If the array `A` is shared no compiler-generated communication will be necessary for updating the values of A.

If the array `B` is shared no compiler-generated communication will be necessary for accessing the values of B.

12.6 PURE Procedures

Pure subroutines and pure functions can be used within parallel loops.

```
      PURE REAL FUNCTION f (x1, x2)
      REAL x1, x2
      f = (x1 - 1) * (x2 + 1)
      END

      REAL a(n,m), ra(n,m)
      INTEGER n, m
!hpf$ distribute ra(*,*)
      FORALL (i=1:n,j=1:m)
         a(i,j) = f(a(i,j), ra(i,j))
      END FORALL


      PURE SUBROUTINE s(i, x)
      INTEGER i
      REAL x
      ...
      END

      PROGRAM p
      REAL a(n)
!hpf$ distribute a(block)
      ...
!hpf$ INDEPENDENT, RESIDENT, ON HOME a(i)
      DO i = 1, n
         a(i) = 1.0
         CALL s(i, a(i))
      END DO
```

It is possible to call pure subprograms with replicated data. But an update is done only on the local incarnation of the variable.

Access to local data will cause no problems in a pure subprogram. The following kind of application with a pure subprogram is quite useful:

```
      PURE SUBROUTINE p (i)
      INTEGER i
      COMMON /yom/ a
      REAL a(100)
!hpf$ distribute a(block)
      REAL x
      x = a(i) + 1.0       ! no broadcast of a(i) required
      a(i) = x             ! a(i) is local data by assertion
      END
```

By this way it is possible to work independently on local data of a common block.

12.7 Local Procedures

A local routine is called on every processor and has by default no communication in it. A serial routine is called on a single processor and has no communication.

Nevertheless it might be possible to call message passing commands within a local routine, but in this case the user himself is responsible for any communication. But only local data can be accessed within the local routine.

Local routines can also be used to provide an HPF interface to implementation-specific parallel libraries.

If one processor is executing a local routine, it might be possible that only some processors are executing this incarnation. Therefore in a local routine it is not possible to have any global operation like redistribution or calling global routines. So we have the following restrictions for local routines:

- Within a local routine only other local routines can be called. It is not possible to call any global routine.
- Every mapping directive is considered as a descriptive directive. There can be never any redistribution within a local routine.

As already mentioned, the user can call any message passing code within local routines. But then he himself is responsible that the routine is executed by all participating processors.

```
      PROGRAM WORK
      REAL A(N,N)
!hpf$ distribute A(*,block)
      ...
      CALL SUB (A,N)
      ...
      END
```

12.8 Private Variables

A *private variable* is a variable that has an incarnation on every processor where every processor can modify this variable for his purposes. Replicated variables have also an incarnation on every processor, but the compiler takes responsibility that all processors have always the same value. All incarnations of replicated variables must be *consistent*. This is not the case for private variables, its use requires no communication at all.

Within the context of HPF, private variables will exist in the following situations:

- A variable specified in the NEW option of the INDEPENDENT directive is always private.

```
!hpf$ INDEPENDENT, NEW (s)
      DO I = 1, n
         s = ...           ! no consistency for s required
         x(i) + x(i) * s
      END DO
```

The compiler has not to make sure that after the termination of the loop all processors will have the same value of s.

- All local variables in a PURE procedure are private.

```
      PURE FUNCTION ITERATE (I,J)
      REAL X, Y         ! private variables X, Y
      INTEGER K         ! private variable  K
      X = I * 0.01
      Y = J * 0.02
      K = 3
      ....
      END FUNCTION
```

- All local variables in a LOCAL procedure are private.

Computations that work only on private variables will not imply any communication. The computations are always local.

13 Global Communications

13.1 Broadcast

Every update of a scalar variable is done in such a way that every processor gets afterwards the actual values.

```
      S = S + 1       ! done by all processors

      S = A(I)       ! implies a broadcast

      READ *, S      ! implies a broadcast from the I/O node
```

These kind of updates are also done for replicated arrays.

13.2 Spreading

There are no problems for using the SPREAD function as long as the data is replicated along the dimension over which the data is spread.

```
      REAL a(n), b(m,n), g
!hpf$ distribute b(*,block)
!hpf$ distribute a(block)
      ...
      b = SPREAD (SPREAD(g,1,n),1,m)
      b = SPREAD (a,1,m)
```

In all other situations, the current restrictions for temporary arrays apply also for spread.

```
      REAL b(m,n), a(n)
!hpf$ distribute (CYCLIC,block)
!hpf$ align a(i) with b(*,i)
      b = SPREAD (a,1,m)
      b = SPREAD (b(j,:),1,m)          ! spread of a column
      a = b(j,:) ; b = SPREAD (a,1,m)
```

13.3 Reduction Functions

The following reduction functions are supported by ADAPTOR: `ALL`, `ANY`, `COUNT`, `IALL`, `IANY`, `IPARITY`, `SUM`, `PRODUCT`, `PARITY`, `MINVAL`, `MAXVAL`, `MINLOC` and `MAXLOC`.

If the functions are used without a "dim" argument, the results must be assigned to a replicated variable.

```
      PARAMETER (n=100)
      REAL a(n), s
!hpf$ distribute (...) a
      INTEGER ijk(3)
!hpf$ distribute ijk(*)
      s = sum (a)
      s = minval (a, a .gt. 0.0)
      s = product (a(5:10))
      ijk = minloc (a)
```

The reduction functions can be used with a "dim" argument, but in this case the value must be known at compile time. ADAPTOR will create internally a temporary array that is replicated along the reduction dimension.

```
      PARAMETER (n=100)
      REAL a(n,n), row(n), col(n)
!hpf$ distribute A (...,...)
!hpf$ align row(j) with a(*,j)
!hpf$ align col(i) with a(i,*)

      row = sum (a, dim = 1)
      col = sum (a, dim = 2)
```

13.4 Reduction Operations in Independent Loops

For the global reduction functions of Fortran 90, it is necessary to collect the results of a parallel loop in a temporary array before. This array must have a size equal to the number of loop iterations.

```
      PARAMTER (N=1000000)
      REAL xa(N), x
      ...
      FORALL (i=1:N) xa(i) = complicated_function(i)
      x = sum (xa)
```

As this temporary array may be excessively large, a reduction feature has been proposed for HPF 2.0.

```
      x = 0.0
!hpf$INDEPENDENT, REDUCTION (x)
      DO i = 1, 1000000
         x = x + complicated_function (i)
      END DO
```

ADAPTOR implements this reduction variables as follows: on entry to an independent loop, every processors has its own incarnation of the reduction variable (same type, same shape) associated with each variable in the reduction clause on the `INDEPENDENT` directive. Its initial value is the same value as it has before the entry of the loop. Each processor performs a subset of the loop iterations; when it encounters a reduction statement, it updates its own copy of the reduction variable. A processor is free to perform its loop iterations in any order.

The final value of the reduction variable is computed by combining the local values with the value of the global reduction variable on entry to the loop. The combining is done in the same way as for the Fortran 90 reduction functions.

Attention: ADAPTOR does not create a new local reduction variable (same type, same shape). It does not initialize it to the identity element for the reduction operator at the entry of the loop.

13.5 Simple Reductions

Beside the REDUCTION directive, it is possible to use also the REDUCE statement. Due to the fact, that this is not standardized, the use of the directive is recommended.

```
      REDUCE (fn, var, exp)
```

At first, every processor makes the reduction for its own local iterations (local reduction). The reduction variable must be a replicated variable. After finishing all iterations a global reduction between all node processes is executed, the global value of the reduction is available in the reduction variable on all nodes.

```
      REAL a(n)
!hpf$ distribute a(block)
      ...
      s = 0.0
!hpf$ INDEPENDENT, RESIDENT, ON HOME a(i)
      DO i=1,n
         REDUCE (SUM, s, a(i))
      END DO
```

Furthermore an array variable within the reduction expression is also considered to find out the home of the iterations of a local independent loop.

The following reduction functions are available:

- COUNT for logical values
- SUM, PRODUCT for integer, real or complex values
- ANY, ALL, PARITY for logical values
- IALL, IANY, IPARITY for integer values
- MINVAL, MAXVAL for real or integer values

13.6 Position Reductions

With the previous reductions it is impossible to determine the position of a minimum or maximum. This can be done with some additional parameters in the REDUCE statement.

```
   REDUCE (pos_reduction, red_variable, red_expression,
           pos_var1, pos_exp1,
           ...
           pos_varn, pos_expn)
```

The semantic of this loop is that the position variables (replicated variables) will have the values of the position expressions of the iteration where the minimum or maximum value has been found.

The following parallel loop determines the minimum value and its position in the two-dimensional array B.

```
      REAL b(n,n), min
!hpf$ distribute b(*,block)
      INTEGER i1, i2, imin1, imin2
      ...
!hpf$ INDEPENDENT, RESIDENT
      DO i2 = 1, n
         DO i1 = 1, n
            REDUCE (MINVAL, min, b(i1,i2), imin1, i1, imin2, i2)
         END DO
      END DO
```

14 Structured Communication

Structured communication will be generated for all data parallel statements where every processor can compute by its own the corresponding schedule for the communication. The schedule specifies which data has to be sent to other processors and which data has to be received.

14.1 Assignments with Regular Sections

A regular section of a distributed array can be assigned to another regular section of any other distributed array. If this assignment needs communication, this will be usually very fast.

```
      REAL a(n,n), a1(n,n)
!hpf$ distribute a(*,block) :: a, a1
      REAL g

      a(1:n,1) = a(1:n,2)     ! fast communication

      a(1:n-1,1:n-1) = a(2:n,2:n)      ! fast communication
      a(3:n,1:n-2)   = a(1:n-2,3:n)    ! fast communication
```

The following examples show that this kind of assignment can also be used to replicate data.

```
      REAL a(n,n), ra(n,n), ra1(n)
!hpf$ distribute (CYCLIC,block) :: a
!hpf$ distribute (*)            :: ra1
!hpf$ distribute (*,*)          :: ra
      INTEGER k
      REAL g

      ra = a
      ra(2:n,1:n-1) = a(1:n-1,2:n)

      ra1 = a(k,1:n)  ! replication of the k-th row of a
      ra1 = a(1:n,k)  ! replication of the k-th column of a
```

Furthermore, such an assignment can imply the redistribution of a whole array.

```
      integer N1, N2, N3, N4, N5, N6
      parameter (N1=7, N2=9, N3=12, N4=5, N5=5, N6=4)

      real A (N1, N2, N3, N4, N5, N6)
      real B (N1, N2, N3, N4, N5, N6)

!hpf$ distribute A (block,*,block,*,*,*)
!hpf$ distribute B (block,CYCLIC,*,*,*,CYCLIC)
      ...
      A = B      ! redistribution of an entire array
      ...
      B(5,2:4,3:7,4,:,:) = A(4,2:4,2:6,5,:,:)  ! same for subsections
```

14.2 FORALL Statements with Structured Communication

A `FORALL` statement might require communication between the available processors.

```
      REAL a(n), b(n)
!hpf$ distribute (block) :: a, b
      ...
      FORALL (i=2:n-1)
         a(i) = (b(i+1) + b(i-1) + 2. * b(i)) * .25
      END FORALL
```

14.3 Shifting

ADAPTOR supports the intrinsic function `CSHIFT` only for whole arrays. Source and target array must have the same shape and the same distribution. Circular shifting is not possible for arrays with a cyclic distributed dimension.

```
      REAL b(m,n), g
!hpf$ distribute b(*,block)     ! b is distributed
      ...
      b = CSHIFT (b,1,1)        ! no communication
      b = CSHIFT (b,2,-1)       ! efficient communication
```

The intrinsic function `EOSHIFT` is supported in the same way.

14.4 Transpose

The intrinsic function `TRANSPOSE` causes no problems at all and will be handled in the same efficient way if communication is involved.

```
      REAL a(m,n), b(n,m)
!hpf$ distribute (*,block) :: a, b
      ...
      a = TRANSPOSE (b)
      a(2:m,1:n-1) = TRANSPOSE (b(2:n,1:m-1))
```

14.5 Matrix Multiplication

ADAPTOR will use a parallel implementation of the matrix multiplication for the array intrinsic function `MATMUL`.

14.6 Shadow Edges

Many scientific application contain a lot of so-called stencil operations where for the update of one element only values of the direct neighborhood is needed. Shadow edges (or overlap areas) that can contain these values guarantee that for the corresponding array statements or parallel loops it is not necessary to create temporary data. Instead of the movement of data to the temporary it is only necessary to update the overlap area.

The following HPF example program shows the benefit of shadow edges.

```
      REAL F(N,N), DF(N,N)
!hpf$ distribute F (block,block)
!hpf$ align DF with F
      ...
      DF (2:N,1:N-1) = F (2:N,2:N) + F (1:N-1,1:N-1)
! equivalent to
      FORALL (J=1:N-1,I=2:N) DF(I,J) = F(I,J+1) + F(I-1,J)
```

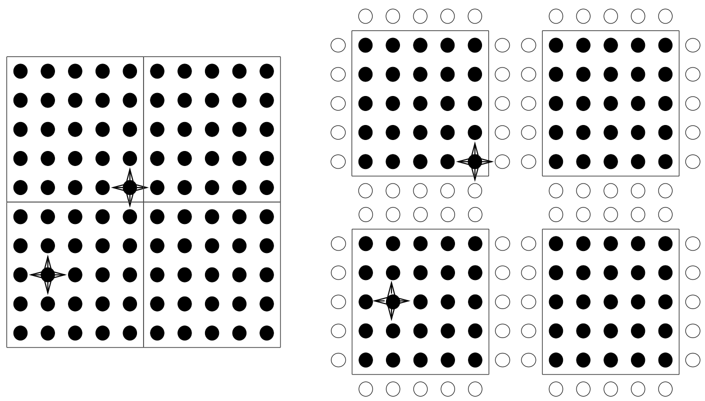
The array assignment requires data movement. Though the data movement contains structured communication, a straight-forward translation would generate the following code:

```
      REAL*4 F (1:N,1:N)
      REAL*4 DF (1:N,1:N)
      REAL*4, ALLOCATABLE :: TMP1 (:,:), TMP2 (:,:)
!hpf$ distribute F (block,block)
!hpf$ align with F(I,J) :: DF(I,J), TMP1(I,J), TMP2(I,J)
      ...
      ALLOCATE (TMP1(1:N,1:N), TMP2(1:N,1:N)
      TMP1(2:N,1:N-1) = F(2:N,2:N)        ! movement
      TMP2(2:N,1:N-1) = F(1:N-1,1:N-1)    ! movement
      DF(2:N,1:N-1) = TMP1(2:N,1:N-1)+TMP2(2:N,1:N-1)  ! local
      DEALLOCATE (TMP2, TMP1)
```

This solution requires two temporary arrays. The two array movements imply some communication between the neighbored processors. Nevertheless a lot of local data will be copied as most neighbored data is on the processor itself. By utilizing shadow edges the following code will be generated:

```
      REAL*4 F (1:N,1:N) ovlp (1:0,0:1)
      REAL*4 DF (1:N,1:N)
!hpf$ distribute F (block,block)
!hpf$ align DF (I_1,I_2) with F(I_1,I_2)
      OVERLAP Update F BY [0:0,0:1]
      OVERLAP Update F BY [1:0,0:0]
      DF(2:N,1:N-1) = F(2:N,2:N)+F(1:N-1,1:N-1)  ! local
```

Though the updating of the shadow edeges area requires the same amount of communication as the assignments to the temporaries, the benefit is due to the fact that no copying of local data is required. Furthermore, the need of memory for the overlap area is less than for the temporary arrays.



(a) global view **(b) local view**

Figure 13: Data partitioning with shadow edges

In contrary to the latest version overlap areas are detected automatically. Nevertheless, the user has still the possibility to specify a certain size for the overlap area. The overlap area will not change the semantic of any program but can increase the performance of a program dramatically.

```
      REAL A(N,N), B(N,N)
!ADP$ OVERLAP B(1:1,1:1)
```

ADAPTOR will use overlap areas also for aligned arrays.

```
      SUBROUTINE RESTR (NPC, NPF, FC, UF, FF)
      INTEGER NPC, NPF
      DOUBLE PRECISION FC (NPC, NPC), FF (NPF, NPF), UF (NPF, NPF)
CHPF$ distribute FF (block, block)
CHPF$ align UF (I,J) with FF (I,J)
CHPF$ align FC (I,J) with FF (2*I-1,2*J-1)

      INTEGER NC

      NC = NPC - 1
      FC (2 : NC, 2 : NC) =
     1      2 * (FF (3 : 2 * NC - 1 : 2, 3 : 2 * NC - 1 : 2) -
     2      4.0D0 * UF (3 : 2 * NC - 1 : 2, 3 : 2 * NC - 1 : 2) +
     3              UF (3 : 2 * NC - 1 : 2, 2 : 2 * NC - 2 : 2) +
     4              UF (2 : 2 * NC - 2 : 2, 3 : 2 * NC - 1 : 2) +
     5              UF (4 : 2 * NC : 2, 3 : 2 * NC - 1 : 2) +
     6              UF (3 : 2 * NC - 1 : 2, 4 : 2 * NC : 2))
      END
```

15 Explicit and Implicit Redistributions

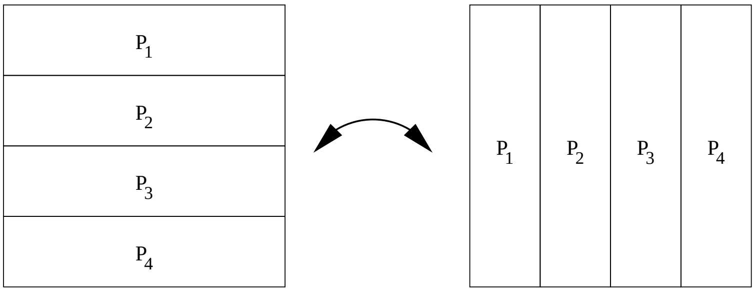
15.1 Explicit Redistributions

The following code contains two loop nests. In the first loop nest, every row can be computed independently, In the second loop nest, every column can be computed independently.

```
      REAL, DIMENSION (N,N) :: A, B
!hpf$ distribute (block,*) :: A, B
      DO J = 2, N
         FORALL (I=1:N)
     &      A(I,J) = A(I,J) - A(I,J-1) * B(I,J)
      END DO
      DO I = 2, N
         FORALL (J=1:N)
     &      A(I,J) = A(I,J) - A(I-1,J) * B(I,J)
      END DO
```

In the `DO J` loop, the `FORALL` statement is local and requires no communication. The `DO I` loop contains also a `FORALL` statement, but it will be executed serially as it is related to a serial dimension.

If we transpose the arrays `A` and `B` before the `DO I` loop (see figure 14, then this loop can be executed exactly as the `DO J` loop.



(a) DISTRIBUTE A(BLOCK,*) (b) DISTRIBUTE A(*,BLOCK)

Figure 14: Redistribution of arrays

```
      REAL, DIMENSION (N,N) :: A, B, A1, B1
!hpf$ distribute (block,*) :: A, B
!hpf$ distribute (*,block) :: A1, B1
      ...
      DO J = 2, N
         FORALL (I=1:N)
     &      A(I,J) = A(I,J) - A(I,J-1) * B(I,J)
      END DO
      A1 = A; B1 = B      ! redistribution
      DO I = 2, N
         FORALL (J=1:N)
     &      A1(I,J) = A1(I,J) - A1(I-1,J) * B1(I,J)
      END DO
      A = A1; B = B1      ! redistribution
```

```
      REAL, DIMENSION (N,N) :: A, B
      ...
!hpf$ REDISTRIBUTE (block,*) :: A, B
      DO J = 2, N
         FORALL (I=1:N)
     &      A(I,J) = A(I,J) - A(I,J-1) * B(I,J)
      END DO
!hpf$ REDISTRIBUTE (*,block) :: A, B
```

```
 DO I = 2, N
    FORALL (J=1:N)
&     A(I,J) = A(I,J) - A(I-1,J) * B(I,J)
 END DO
```

15.2 Redistributions in the Called Routine

With the exception of local routines, serial routines and pure procedures, every subprogram will check the distributions of its dummy arguments and make some redistribution if it is necessary.

`Attention:` Also descriptive directives will be handled like prescriptive ones. Inherited distributions are not supported.

As the subroutine is responsible for the redistribution, the user can take advantage of the `INTENT` attribute. It can avoid the copy in or copy out of data in case of a redistribution.

15.3 Redistributions in the Calling Routines

In the following it will be explained in which situations it is useful, it might be useful and it is necessary to have an interface block.

The most general rule is that any full array or any section of an array is just passed by a descriptor and the called subroutine is responsible for a redistribution. The calling routine has not to do anything and therefore no interface block is necessary.

```
call SUB (A(1:N,1:N), B)
```

The next general rule is that an interface block must be available if the subroutine will not apply redistributions and cannot deal with the actual distribution. In the following cases a subroutine cannot apply redistributions:

- A serial procedure is only called by a single processor (see also section 9.2). An interface block must be available in any case.
- Local routines will not redistribute their dummy arguments (see also section 9.1). An interface block must be available if it is called with any other distribution than the expected one.

The last rule is that an interface should be specified if the calling routine should make the redistribution for optimization issues. This may occur if

- the actual argument requires a temporary array in any case,
- or if the routine is called within a loop and the redistribution must be done during every iteration.

Note: About a redistribution within the called routine will be decided at runtime. If the interface block is available in the calling routine, the redistribution will be decided at compile time. In some situations, it can only be determined at runtime that no redistribution is necessary, e.g. on one processor the block and cyclic distribution is the same. Then at least the local copying of data is not necessary.

16 Unstructured Communication

In contrary to the structured communication, there is also communication necessary to set up the communication schedule. This communication is needed to ask for the needed data or to inform the processor about data that will be sent.

16.1 Gathering of Data

The following code shows a very convenient possibility of gathering data from a distributed array.

```
      type B(n1,n2,...,nk)
      type A(m1,...,mk)
      logical MASK (m1,...,mk)
!hpf$ distribute A (...)
!hpf$ distribute B (...)
      INTEGER P1(m1,...,mk), ..., Pn(m1,...,mk)
!hpf$ align (i1,...,ik) with A(i1,...,ik) :: P1, ..., Pn, MASK
      ....
      FORALL (j1=low1:up1, ..., jk = lowk:upk, MASK(j1,...,jk))
     &   A(j1,j2,...,jk) = B(P1(j1,...,jk), ..., Pn(j1,...,jk))
```

ADAPTOR will generate rather efficient parallel code if the following points are observed:

- P1, ..., Pn must be integer arrays, where n is the rank of the array B. The values of these arrays must be legal index values for the corresponding index of B.
- A, P1, ..., Pn and MASK should have the same rank, the same shape and should be aligned with each other.
- A and B should be of the same type, as implicit type conversion might decrease the efficiency.

The indirect addressing of a distributed array requires complex runtime support. A communication pattern must be computed to access the needed data. In a first step, every processor will ask the other processors for the needed data. In the second step, the processors send the required values to the corresponding processors. This functionality must be available in the runtime system and has usually a very high overhead.

In case of a shared array this runtime support is not necessary. Especially the owner evaluation is no longer necessary as all global addresses remain unchanged.

```
      REAL A(N), B(M)
      INTEGER IND(N)
!hpf$ distribute (block) :: A, B, IND
!ADP$ SHARED B
      ...
      A = B(IND)
```

The arrays A and IND are aligned. Every processor will need some values of the array B. As this data is shared, no communication has to be inserted by the compiler.

16.2 Scattering of Data

The following code shows a very convenient possibility of scattering data to a distributed array.

```
      type B(n1,n2,...,nk)
      type A(m1,...,mk)
      logical MASK (m1,...,mk)
!hpf$ distribute A (...)
!hpf$ distribute B (...)
      INTEGER P1(m1,...,mk), ..., Pn(m1,...,mk)
!hpf$ align (i1,...,ik) with B(i1,...,ik) :: P1, ..., Pn, MASK
      ....
      FORALL (j1=low1:up1, ..., jk = lowk:upk, MASK(j1,...,jk))
     &   B(P1(j1,...,jk), ..., Pn(j1,...,jk)) = A(j1,j2,...,jk)
```

The HPF intrinsic functions should be used to scatter data from an array A to an array B:

```
B = xxx_SCATTER (A, B, P1, ..., Pn, A, MASK)
```

The MASK parameter is optional. The allowed values for the reduction function are ALL, ANY, COUNT, IALL, IANY, IPARITY, SUM, PRODUCT, PARITY, MINVAL and MAXVAL.

The following has to be observed:

- P1, ..., Pn must be integer arrays, where n is the rank of the array B. The values of these arrays must be legal index values for the corresponding index of B
- A, P1, ..., Pn and MASK must have the same rank, the same shape and the same distribution.
- A and B must be of the same type, e.g. no implicit type conversion is done here.

If k is the rank of the arrays A, P1, ..., Pn and MASK, and (low1:up1,...,lowk:upk) the shape, the semantic of the scatter operation can be described by the following loop nesting:

```
 DO j1 = low1, up1
  DO j2 = low2, up2
   ...
    DO jk = lowk, upk
      IF (MASK(j1,...,jk) THEN
        B(P1(j1,...,jk), ..., Pn(j1,...,jk)) =
&         red_f (B(P1(j1,...,jk), ..., Pn(j1,...,jk)), A(j1,...,jk))
      END IF
    END DO
   ...
  END DO
 END DO
```

16.3 Indirect Addressing

The direct use of indirect addressing is only supported for full arrays.

```
REAL a(n), b(m)           ! arrays are all
INTEGER p(n)              ! distributed by default

b(p) = a                  ! scatter operation

a = b(p)                  ! gather operation
```

It should be mentioned that in many situations indirect addressing can be translated to corresponding calls of the scatter routines.

```
dimension Y(km), Z(km)
INTEGER INDEX(km)
...
Y(INDEX(1:LIMIT)) = Y(INDEX(1:LIMIT)) + Z(1:LIMIT)
```

This operation can be rewritten to a scatter operation.

```
dimension Y(km), Z(km)
INTEGER INDEX(km)
...
Y = SUM_SCATTER (Z(1:LIMIT), Y, INDEX(1:LIMIT), mask)
```

The following example demonstrates how to handle code with two steps of indirect addressing.

Attention: Indirect addressing within a FORALL statement is supported. But as indirect addressing might involve a lot of communication it should be used carefully.

16.4 The TRACE Directive

The requirement for tracing is described by the user directive `!ADP$ TRACE`. This directive has a Fortran attribute semantics, and follows Fortran 90 rules of scoping. By default, arrays are not traced.

An array must be flagged dirty when it is changed. The syntactic constructs that can modify an array are limited, and can be analyzed at compile-time: assignments with this array as a left-hand-side, redistributions, allocations and deallocations. Thus, the trace attribute is compatible with the `DYNAMIC` and `ALLOCATABLE` attributes, but not with the `POINTER` or `TARGET` attribute, because the compiler will not be able to record modification of such arrays.

```
       integer, dimension (N) :: P
!adp$  trace :: P
```

Communication schedules can be reused if indirection array has not changed. In the following example, the communication schedule will be reused for the integer array `P` as it is traced and not modified between the different calls of the subroutine `TIMING`.

```
      subroutine INDIRECT
      real, dimension (N)     ::  A, B
      integer, diemension (N) :: P, Q
!hpf$ distribute (block) :: A
!hpf$ align with A :: B, P, Q
!adp$ trace P
      call init(P)
      call init(Q)
      do J = 1, M ! serial loop
         call timing (A, B, P)
      end do
      do J = 1, M ! serial loop
         call timing (A, B, Q)
      end do
      end subroutine indirect

      subroutine timing(A, B, L)
      ...
!adp$ trace L
      forall (I=1:N) A(I) = B(L(I))
      end subroutine timing
```

17 Extracting Communication

17.1 Temporary Arrays

Complex array statements and parallel loops might contain a lot of communication. The general strategy is to extract the communication outside of the parallel loop to get local independent computations. For this process, it might be necessary to introduce temporary arrays.

```
      integer, parameter :: N=100, NA=200, NB=50

      real, dimension (N)  :: VAL
      real, dimension (NA) :: A
      real, dimension (NB) :: B
!hpf$ distribute (block) :: VAL, A, B

      INTEGER IND1(N), IND2(N)
!hpf$ align(I) with VAL(I) :: IND1, IND2

      FORALL (I=1:N) A(IND1(I)) = B(IND2(I)) + VAL(I)  ! on home VAL(I)
```

The complex `FORALL` statement will be handled in the following way:

```
      REAL TMP1(N), TMP2(N)
!hpf$ align(I) with VAL(I) :: TMP1, TMP2

      FORALL (I=1:N) TMP1(I) = B(IND2(I))         !  pre-fetch

      FORALL (I=1:N) TMP2(I) = TMP1(I) + VAL(I)   !  local

      FORALL (I=1:N) A(IND(I)) = TMP2(I)          !  post-store
```

Before the parallel loop, which itself is now a local one, the processors collect the data that will be needed within the parallel loop. This is also called *pre-fetching*.

After the parallel loop, the processors will send the non-local computed data to the owner of the data. This is also called *post-store*. Due to the owner-computes rule, in most situations this is not necessary. But in case of indirect-addressing or more complex parallel loops, this kind of communication must be generated.

17.2 Problems with Extracting Communication

Some situations can be identified where it is not easy or not possible to extract the communication completely outside of the parallel loop.

- In the following loop, communication might be necessary to have local copies of the values `X(I+k)`:

```
                                        TMP(1:N) = X(k+1:k+N)
!hpf$ INDEPENDENT                 !hpf$ INDEPENDENT, RESIDENT ON HOME Y(I)
      DO I = 1, N                       DO I = 1, N
         X(I) = ...                        X(I) = ...
         Y(I) = f(X(I+k))                  Y(I) = f(TMP(I))
      END DO                            END DO
```

  But extracting the communication as it is done here produces wrong code if the value of `k` is 0. In this case, we have loop independent true dependences, as the needed value for X is computed during the same iteration.

- Extracting the communication is not possible for a subroutine call as long as it is not known which non-local data will be accessed within the subroutine.

```
!hpf$ INDEPENDENT
      DO I = 1, N
         X(I) = ...
         CALL SUB (I, X(I), Y(I))
      END DO
```

- Though the subroutine might be a `PURE` subroutine, it is possible to read values of global arrays within the subroutine. If such an array is distributed, communication might be involved that cannot be extracted.

```
      PURE SUBROUTINE f (x1)
      REAL x1
      INTEGER p
      p = g(x1)
      x1 = x1 * a[p]
      END
```

The extraction of communication is absolutely necessary as every processor must be involved who has to send data or who has to receive data.

This problem does no longer exist, if one-sided communication is available. In this case, one processor can access non-local data during the parallel execution. Only synchronization before and after the parallel loop is necessary.

Appendix: Fortran 90

This appendix gives a short summary of new features in Fortran 90. It was adopted in 1991 and is now an ANSI and the ISO standard.

17.3 Syntax Improvements

- new declaration statements,
- the binary operations `<>`, `/=`, `==`, `<=`, `<`, `>`, and `>=` instead of `.ne.`, `.eq.`, `.le.`, `.lt.`, `.gt.`, and `.ge.`,
- ending comments starting with `!`,
- semicolon `;` for separating statements,
- using `&` for continuation lines,
- free source format.

17.4 Dynamic Arrays

Fortran 90 supports two kinds of dynamic arrays:

- *Allocatable arrays* - explicit allocation (by allocate statement) and deallocation (by deallocate statement)
- *Automatic arrays* - automatic allocation (upon entry to the defining subprogram) and deallocation (on return).

While the allocatable arrays use heap storage, automatic arrays can use stack storage.

17.4.1 Allocatable Arrays

An allocatable array is always local (it cannot be a dummy argument or be declared in common). It can be allocated and deallocated only locally. This kind of array will be used if user input specifies the size of the arrays at runtime.

```
REAL, ALLOCATABLE :: a(:), b(:)
READ *, n
IF (n .GT. 0) THEN
   ALLOCATE (a(n), b(n))
   ...
   DEALLOCATE (b, a)
END IF
```

17.4.2 Automatic Arrays

An automatic array can appear only in a subprogram. It looks similar to a static array but the bounds are specified as dummy arguments or elements of a common block. In any case, an automatic array is not a dummy array and not part of a common block.

```
SUBROUTINE s (n)
REAL a(n), b(n)
...
a(2:n) = b(1:n-1)
```

17.5 Array Syntax

Array syntax allows to specify operations on full arrays or on sections of them.

```
real, dimension(1:100,1:100) :: A, B, C     ! Declare Arrays
real, dimension(1:50)         :: X, Y        ! Declare vectors
A = 1.0                        ! assign whole array
B = 2.0
C = A * B                      ! combine whole arrays
X = C(1,1:100:2)               ! pick alternate elements from row 1
Y(1:50:-1) = X(1:50)           ! reverse order of vector
```

The `WHERE` construct works as a masking operation for array statements. It can be used as a single statement thus:

```
WHERE (A >= 0.0) A = SQRT (A)
```

or as a block `WHERE` construct such as

```
WHERE (mask)
   A = SQRT (A)
 ELSE WHERE
   A = default
END WHERE
```

An array value can be formed with an array constructor (is always a one-dimensional array).

- an array constructor is `'['index_list']'` (ADAPTOR supports `'['` and `']'` as well as `'(/'` and `'/)'`).
- an index can be a single element, a range specifier or an implied do loop.

```
parameter (n=6)
real a(n)
...
A = (/(i=1,n)/)
A = [1:n]
A = [1,2,3,4,5,6]
A = (/ 2, 3, 4, (I, I = 13, 43, 5) /)   ! not supported
A = [2, 3, 4, [13:43:5]]                ! not supported
```

17.6 Array-valued Functions

With Fortran 90, functions can return full arrays instead of only scalar variables.

```
function iota (n)
integer n
integer iota (n)
iota = (/(i,i=1,n)/)
end
```

17.7 Assumed-Shaped Arrays

A dummy array is an assumed-shaped array if there are no declared bounds for it. The bounds will be paased in a data descriptor for the array. Intrinsic functions are provided to query the bounds.

```
 subroutine init (A)
 real A(:,:)
 do j = lbound(A,2), ubound(A,2)
   do i = lbound(A,1), ubound(A,1)
      A(i,j) = 1.0
   end do
 end do
```

17.8 New Control Structures

The new control constructs of Fortran 90 can be used with ADAPTOR. They will be translated to equivalent FORTRAN 77 constructs.

- The `CASE` construct,
- the `EXIT` statement,
- the `CYCLE` statement,
- and `DO` loops without loop control.

```
do
   read *, number
   print *, "input data: ", number
   if (number < 0) then
      exit
   else if (mod(number, 2) == 0) then
      cycle
   else
      number_of_odd_numbers = number_of_odd_numbers + 1
   end if
end do
```

```
print *, "enter traffic_light color"
read *, traffic_light
select case(traffic_light)
case ("red")
print *, "stop"
case ("yellow")
print *, "caution"
case ("green")
print *, "go"
case default
print *, "illegal value:", traffic_light
end select
```

17.9 Parameterized Data Types

Portability of numerical code has long been difficult, primarily due to differences in the word sizes of the computers on which the code is run. Fortran 90 introduces parameterized types, increasing portability of software from machine to machine. This is done using *kind values*, constants associated with an intrinsic type such as `integer` or `real`. Parameterization of kind values allows precision changes by changing a single constant in the program. Several intrinsic functions are provided to select kind values based on the range and precision desired and inquire about a variable's precision characteristics in a portable way.

```
module Precision
   integer, parameter :: Q = selected_real_kind( 10, 10 )
end module Precision

program Portable
  real (kind=Q) :: a, b, c
  ...
end program Portable
```

The `selected_real_kind` function above selects the kind value corresponding to a real number with at least 10 decimal digits of precision and a decimal exponent range of at least 10 in magnitude. The `selected_int_kind` function is similar, and an expression such as `selected_int_kind(10)` selects the kind value corresponding to a integer number with magnitude in the range $(10^{-10}, 10^{10})$.

Function	Description
`digits(x)`	q for an integer argument, p for a real argument
`epsilon(x)`	b^{1-p} for a real argument
`huge(x)`	Largest in the integer or real model
`minexponent(x)`	Minimum value of e in the real model
`maxexponent(x)`	Maximum value of e in the real model
`precision(x)`	Decimal precision (real or complex)
`radix(x)`	The base b of the integer or real model
`range(x)`	Decimal exponent range (real, complex, or integer)
`tiny(x)`	Smallest positive value in the real model

Table 2: Numeric Inquiry Functions

17.10 Numerical Inquiry and Manipulation Functions

Fortran 90 introduces several intrinsic functions to inquire about machine dependent characteristics of an integer or real. For example, the inquiry function, `huge`, can be used to find the largest machine representable number for an integer or real value. The integer model used by these inquiry functions is[1]

$$i = s \sum_{k=0}^{q-1} d_k r^k$$

where

- i is the integer value
- s is the sign (+1 or -1)
- r is the radix ($r > 1$)
- q is the number of digits ($q > 0$)
- d_k is the kth digit, $0 \leq d_k < r$.

The floating-point model used by the inquiry functions is

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}$$

where

- x is the real value
- s is the sign (+1 or -1)
- b is the base ($b > 1$)
- e is the exponent
- p is the number of mantissa digits ($p > 1$)
- f_k is the kth digit, $0 \leq f_k < b$, $f_1 = 0 \Rightarrow f_k = 0 \,\forall\, k$.

Table 2 lists intrinsic functions that inquire about the numerical environment. Table 3 lists intrinsic functions that manipulate the numerical characteristics of variables in the real model. An important feature of all of these intrinsic functions is that they are generic and may be used to obtain information about any *kind* of integer or real supported by the Fortran 90 implementation.

ADAPTOR supports the numeric inquiry and the numeric manipulation functions. The use of the numeric manipulation functions will require a Fortran 90 compiler on the target machine.

[1]Information on the integer and floating-point models, as well as the following tables is taken from chapter 13 of [ABM+92].

Function	Description
`exponent(x)`	Value of e in the real model
`fraction(x)`	Fractional part in the real model
`nearest(x)`	Nearest processor number in a given direction
`rrspacing(x)`	Reciprocal of relative spacing near argument
`set_exponent(x)`	Set the value of e to a specified value
`spacing(x)`	Model absolute spacing near the argument

Table 3: Numeric Manipulation Functions

17.11 Interface Blocks

In FORTRAN 77 interfaces are always implicit. In Fortran 90 interfaces can be explicit:

- in the same compilation unit interfaces are explicit,
- the `USE` statement imports the interfaces of a module (see section 17.15) and makes them explicit,
- the interface can be made explicit by an `INTERFACE` block.

```
INTERFACE
  SUBROUTINE SUB (A, B)
  REAL A(:,:)
  INTEGER, POINTER :: B
  END
END INTERFACE
...
CALL SUB (X, Y)    ! pass descriptor for X instead of pointer
```

Certain uses (such as `POINTER` dummies and assumed-shape array dummies, optional arguments) require an explicit interface.

17.12 Optional Arguments

In Fortran 90 it is possible to indicate that certain arguments to a procedure are optional arguments in the sense that they do not have to be present when the procedure is called. An optional argument must be declared by the `OPTIONAL` attribute.

```
SUBROUTINE DOIT (M, N, S, D)
INTEGER N
REAL S, D
INTEGER, OPTIONAL :: M
...
END SUBROUTINE DOIT
```

In the example the argument `M` needs not to be available. The subroutine can be called with any of the following statements:

```
call DOIT (0, 7000, 0.1, 100.0)
call DOIT (0, 7000, D=0.1, S=100.0)
call DOIT (N=7000, D=0.1, S=100.0)
call DOIT (D=0.1, S=100.0, N=7000)
call DOIT (M=0, N=7000, D=0.1, S=100.0)
```

The presence of an optional argument can be tested with the intrinsic inquiry function `PRESENT`.

Optional arguments can be used within ADAPTOR. It should be observed that for user functions explicit interface blocks should be available if optional arguments are given.

17.13 Derived Data Types

The user can define new data types, created from a collection of intrinsic types. These are similar to the concept of structures or records in other languages.

```
TYPE POINT
   INTEGER :: X, Y
END TYPE
```

Objects of a derived data type can be defined in the following way:

```
TYPE(POINT) :: P1, P2
...
P1%X = 0.0; P1%Y = 0.0
P2   = P1;
```

By overloading existing operators it is possible to define new operations on derived data types (see section 17.18).

17.14 Pointers

In Fortran 90 objects can have the `POINTER` attribute. No storage will be allocated for such an object. The object can be *pointer associated* to an existing object or to an object that will be created with the `ALLOCATE` statement.

```
REAL, POINTER :: P
REAL, TARGET  :: X
...
P => X          ! P is associated with X
P = 5.3
PRINT *, X      ! will print the value 5.3
ALLOCATE (P)    ! P is associated with a new real variable
...
DEALLOCATE (P) ! free memory to which P is associated
```

If a pointer is associated with an existing variable, this variable must have the `TARGET` attribute or must be itself an associated pointer. A pointer can also be an alias to a row or column of an array.

```
REAL, TARGET, DIMENSION (N,N) :: A
REAL, POINTER, DIMENSION (:) :: P
....
P => A(3,:)     ! P is an alias to the third row of A
P => A(:,5)     ! P is an alias to the fifth column of A
```

The `ASSOCIATED` intrinsic function checks whether a pointer is associated with a particular target, or with any target.

A component of a derived type can be a pointer. By this way, it is now possible to have dynamic data structures in Fortran programs.

17.15 Modules

Common blocks in FORTRAN 77 were the only portable means of achieving global access of data throughout a collection of subprograms. This is unsafe, error-prone, and encourages bad programming practices in general. Fortran 90 provides a new program unit, a *module*, that replaces the common block and also provides many other features that allow modularization and data hiding, key concepts in developing large, maintainable numerical code.

Modules consist of a set of declarations and **module procedures** that are grouped under a single global name available for access in any other program unit via the `use` statement. **Interfaces** to the contained module procedures are explicit and permit compile time type-checking in all program units that use the module. Visibility of items in a module may be restricted by using the `private` attribute. The `public` attribute is also available. Those identifiers not declared `private` in a module implicitly have the `public` attribute.

```
module TypicalModule
   private SWAP   ! Make swap visible only within this module.

   contains

   subroutine ORDER (X, Y)   ! Public by default.
      integer, intent( inout ) :: X, Y

      if ( abs( x ) < abs( y ) )  call SWAP (x, y)
   end subroutine order

   subroutine SWAP (X, Y)
      integer, intent( inout ) :: X, Y
      integer TMP

      TMP = X; X = Y; Y = TMP ! Swap X and Y.
   end subroutine SWAP

end module TypicalModule

program UseTypicalModule
   use TypicalModule

   ! Declare and initialize x and y.
   integer :: x = 10, y = 20

   print *, x, y
   call ORDER ( x, y )
   print *, x, y
end program UseTypicalModule
```

A module collects also all interfaces in one place. The `USE` statement imports also the interfaces.

17.16 Internal Procedures

In FORTRAN 77, all subprograms are external with the exception of statement functions. Internal subprograms are now possible under Fortran 90 and achieve an effect similar to FORTRAN 77's statement functions. They are visible only within the containing program and have an explicit interface, guarding against type mismatches in calls to the subprogram. Internal subprograms must be separated from the main program by the `contains` statement. An example illustrating an internal subprogram is given below.

```
PROGRAM LAPLACE

IMPLICIT NONE

! global data

   REAL, ALLOCATABLE :: F(:,:), DF(:,:)
   INTEGER MAXX, MAXY
   ...
   CALL ALLOC_DATA ()
   CALL INIT ()
   ...
```

```
   CALL FREE_DATA ()

CONTAINS

  SUBROUTINE ALLOC_DATA ()
     READ *,MAXX, MAXY
     ALLOCATE (F(MAXX,MAXY), DF(1:MAXX,MAXY))
  END SUBROUTINE ALLOC_DATA

  SUBROUTINE FREE_DATA ()
     DEALLOCATE (DF, F)
  END SUBROUTINE FREE_DATA

  SUBROUTINE INIT ()
     F = 2.
     F(:,MAXY) = 1.
     F(2:MAXX-1,2:MAXY-1) =  0
     DF = 0
  END SUBROUTINE INIT

END PROGRAM LAPLACE
```

When procedures are internal to a program, another procedure, or within a module, they are preceded by a `CONTAINS` statement. The internal procedure must appear just before the last `END` statement of the program, procedure, or module containing them.

17.17 Generic Procedures

```
INTERFACE swap    ! generic name

   SUBROUTINE swap_int (i, j)    ! specific name
   INTEGER i, j
   END SUBROUTINE swap_int

   SUBROUTINE swap_real (X, Y)   ! specific name
   REAL x, y
   END SUBROUTINE swap_int

END INTERFACE
```

17.18 Overloading

Fortran 90 allows to overload existing operators. By this way, it is possible to extend the operations for new defined types.

```
MODULE points
TYPE point
   INTEGER :: x, y
END TYPE
INTERFACE OPERATOR (+)
   MODULE PROCEDURE add_points
END INTERFACE

CONTAINS

TYPE (point) FUNCTION add_points (x, y)
TYPE (point) x, y
...
END FUNCTION add_points
...
END MODULE POINTS


SUBROUTINE s
USE points
TYPE (point) :: p1, p2, p3
...
p3 = p1 + p2
...
END SUBROUTINE s
```

Appendix B: Fortran 95

This appendix gives a short summary of new features in Fortran 95 It was adopted in 1996 and is now an ANSI and the ISO standard.

17.19 The FORALL Statement

The `FORALL` statement can be used for specifying an array assignment in terms of array elements or array sections. It can be masked with a scalar logical expression. The parallelism of this assignment is given by the fact that the assignment can be executed in any order.

The single assignments of a `FORALL` statement will be executed by the owner of the left hand side in the assignment.

It is possible to use array statements within the FORALL loops.

```
      REAL a(n,n), b(n,n), c(n)
!hpf$ distribute (block,block) :: a, b
!hpf$ distribute (block) :: c
      ...
      FORALL (i=1:n) a(1:i,i) = b(1:i,i)
      FORALL (i=2:n) c(i) = sum(c(1:i))
```

17.20 The FORALL Construct

The `FORALL` construct can contain nested `FORALL` statements, `FORALL` constructs and `WHERE` statements. It can be used without any restrictions.

```
      FORALL (I=1:8)
         A(I,I) = SQRT (A(I,I))
         FORALL (J=I-3:I+3, J/=I .AND. J>=1 .AND. J<=9)
            A(I,J) = A(I,I) * A(J,J)
         END FORALL
         WHERE (A(I,:) .NE. 0.0)
            A(I,:) = A(I-1,:) + A(I+1,:)
          ELSEWHERE
            B(I,:) = A(6-I,:)
         END WHERE
      END FORALL
```

In a `FORALL` construct all statements are executed completely in order of appearance.

17.21 PURE Procedures

A pure procedure is designed to guarantee that it is free from side effects (i.e., modifications of data visible outside the procedure). Therefore it is safe to reference it in constructs such as a `FORALL` assignment statement where there is no explicit order of evaluation.

Pure subprograms must have the keyword `PURE`.

```
      PURE REAL FUNCTION f (x1, x2)
      REAL x1, x2
      f = (x1 - 1) * (x2 + 1)
      END

      PURE SUBROUTINE x (a, b, c)
      REAL a, b, c
      c = (a - 1) * (b + 1)
      END
```

A pure procedure must not have any side effects. For this reason, a lot of syntactical restrictions are given for pure routines.

17.22 ELEMENTAL Procedures

Elemental procedures are designed to specify pure routines for scalar arguments that can later also be called with array arguments.

Elemental subprograms must have the keyword `ELEMENTAL`. Every elemental procedure must also be pure, but there are some more restrictions. Dummy procedures are not allowed, all dummy arguments and the function result must be scalar and not have the `POINTER` attribute.

Appendix C: Intrinsics

A lot of Fortran intrinsic functions can be translated with ADAPTOR but still remain Fortran intrinsic functions in the generated code. If they are FORTRAN 77 intrinsic functions they should be compiled by every FORTRAN 77 compiler. But Fortran 90 intrinsic functions might require a Fortran 90 compiler on the target system.

If an intrinsic function is available, it means that ADAPTOR translates it to corresponding FORTRAN 77 code with corresponding support in the runtime system. These functions can be used without any restrictions.

If an intrinsic is noted as restricted, it means that the functionality is not full available, e.g. a parameter value must be known at compile time or an optional parameter must not be available.

If an intrinsic is noted as not available, it means, that it cannot be used with ADAPTOR at all.

17.23 Numeric, mathematical, character, kind, logical and bit procedures

17.23.1 Numeric functions

The numeric functions are `INT`, `REAL`, `DBLE`, `CMPLX`, `AIMAG`, `CONJG`, `AINT`, `ANINT`, `NINT`, `ABS`, `MOD`, `SIGN`, `DIM`, `DPROD`, `MODULO`, `FLOOR`, `CEILING`, `MAX` and `MIN`. They are all elemental functions.

All numeric intrinsic functions are supported. Attention has to be paid to those functions that are not supported by the F77 compiler of the target machine.

Especially there might be problems with the functions `AINT`, `ANINT`, `CMPLX`, `INT`, `NINT` or `REAL` if the `KIND` argument is available.

17.23.2 Mathematical functions

The elemental functions `SQRT`, `EXP`, `LOG`, `LOG10`, `SIN`, `COS`, `TAN`, `ASIN`, `ACOS`, `ATAN`, `ATAN2`, `SINH`, `COSH` and `TANH` evaluate elementary mathematical functions.

They are all supported without any restrictions.

17.23.3 Character functions

The elemental functions `ICHAR`, `CHAR`, `LGE`, `LGT`, `LLE`, `LLT`, `IACHAR`, `ACHAR` are handled.

The elemental functions `LEN_TRIM`, `SCAN`, `ADJUSTL`, `ADJUSTR`, and `VERIFY`, as well as the transformational functions `REPEAT` and `TRIM` are not supported (do not belong to FORTRAN 77).

17.23.4 Character inquiry functions

The inquiry function LEN is supported.

17.23.5 KIND functions

Kind parameters provide are way to parameterize the selection of different possible machine representations for each of the intrinsic data types. This feature is now supported in ADAPTOR. This includes the inquiry function KIND as well as the transformational functions SELECTED_REAL_KIND and SELECTED_INT_KIND.

17.23.6 LOGICAL function

The elemental function LOGICAL is not available.

17.23.7 Bit manipulation and inquiry procedures

Logical operations on bits are provided by the functions IOR, IAND, NOT and IEOR, shift operations are given by the functions ISHFT and ISHFTC. Bit subfields may be referenced by the function IBITS and by the subroutine MVBITS. Single-bit processing is provided by the functions BTEST, IBSET, and IBCLR.

All of these functions are checked by ADAPTOR, but their realization is passed to the compiler of the generated code.

17.24 TRANSFER function

The function TRANSFER is not available.

17.25 Numeric manipulation and inquiry functions

All the corresponding numeric inquiry functions RADIX, DIGITS, MINEXPONENT, MAXEXPONENT, PRECISION, RANGE, HUGE, TINY and EPSILON, and the floating point manipulation functions EXPONENT, SCALE, NEAREST, FRACTION, SET_EXPONENT, SPACING and RRSPACING cannot be used as long as the available Fortran compiler does not support these functions.

17.26 Intrinsic subroutines

17.26.1 Date and time subroutines

The timing routines of Fortran 90 are available to guarantee also a rather good portability of codes that are timed. These routines are DATE_AND_TIME, SYSTEM_CLOCK and CPU_TIME.

```
DATA_AND_TIME ([DATE] [,TIME] [,ZONE], [,VALUES])
SYSTEM_CLOCK ([COUNT] [,COUNT_RATE] [,COUNT_MAX])
CPU_TIME (TIME)
```

17.26.2 Pseudorandom numbers

The intrinsic subroutines RANDOM_NUMBER and RANDOM_SEED are supported by ADAPTOR and realized by own library implementations.

```
RANDOM_NUMBER (harvey)
RANDOM_SEED ([SIZE][,PUT],[,GET])
```

17.27 Array intrinsic functions

17.27.1 Array inquiry functions

The functions LBOUND, UBOUND, SHAPE, SIZE are realized by ADAPTOR and own library functions.

The function ALLOCATED can be used, but the use of allocation is currently restricted (see section 17.4.1).

17.27.2 Vector and matrix multiply functions

The functions DOT_PRODUCT and MATMUL can be used without any restrictions. For both routines parallel code will be generated.

17.27.3 Array reduction functions

The functions ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT and SUM are supported by ADAPTOR. There are efficient realizations for these reductions in the context of distributed arrays.

17.27.4 Array construction functions

The functions MERGE and SPREAD are supported.

The functions PACK and UNPACK can be used but are not parallelized.

17.27.5 Array reshape function

The function RESHAPE can be used but is not parallelized so far.

17.27.6 Array manipulation functions

The array manipulation functions CSHIFT, EOSHIFT and TRANSPOSE are implemented, also rather efficiently for distributed arrays.

17.27.7 Array location functions

The array location functions MINLOC and MAXLOC are available, but cannot be used with the DIM argument.

17.27.8 Pointer association status functions

The functions ASSOCIATED and NULL are not available as ADAPTOR does not support pointer at all.

17.28 HPF Intrinsics and Libraries

The system inquiry functions `NUMBER_OF_PROCESSORS` is supported by ADAPTOR, but not `PROCESSORS_SHAPE`.

The mapping inquiry subroutines `HPF_ALIGNMENT`, `HPF_TEMPLATE`, and `HPF_DISTRIBUTION` are now available in ADAPTOR as this tool now supports also inherited distributions.

The new reduction functions `IALL`, `IANY`, `IPARITY` and `PARITY` can be used with ADAPTOR.

With ADAPTOR, the array combining scatter functions `XXX_SCATTER` are full supported and implemented efficiently.

Array prefix and suffix functions `XXX_PREFIX` and `XXX_SUFFIX` are not supported by ADAPTOR. The same is true for the bit manipulation functions `ILEN`, `LEADZ`, `POPCNT` and `POPPAR` as well as for the sorting functions `GRADE_DOWN` and `GRADE_UP`.

Intrinsic	Available	Class	Type
ABS	F77	Elemental	Numeric
ACHAR	F90	Elemental	Transfer
ACOS	F77	Elemental	Numeric
ADJUSTL	F90	Elemental	Character
ADJUSTR	F90	Elemental	Character
AIMAG	F77	Elemental	Transfer
AINT	F77	Elemental	Transfer
ALL	Yes	Transformational	Array reduction
ALLOCATED	No	Inquiry	Array inquiry
ANINT	F77	Elemental	Transfer
ANY	Yes	Transformational	Array reduction
ASIN	F77	Elemental	Numeric
ASSOCIATED	No	Inquiry	Pointer association
ATAN	F77	Elemental	Numeric
ATAN2	F77	Elemental	Numeric
BIT_SIZE	F90	Inquiry	Bit inquiry
BTEST	F90	Elemental	Bit computation
CEILING	F90	Elemental	Numeric
CHAR	F90	Elemental	Transfer
CMPLX	F77	Elemental	Transfer
CONJG	F77	Elemental	Transfer
COS	F77	Elemental	Numeric
COSH	F77	Elemental	Numeric
COUNT	Yes	Transformational	Array reduction
CPU_TIME	Yes	Subroutine	Subroutine
CSHIFT	Yes	Transformational	Array manipulation
DATE_AND_TIME	Yes	Subroutine	Subroutine
DBLE	F77	Elemental	Transfer

Beside the existing intrinsic routines, the HPF standard describes some new intrinsic routines.

Intrinsic	Available	Class	Type
DIGITS	F90	Inquiry	Numeric inquiry
DIM	F77	Elemental	Numeric
DOT_PRODUCT	Yes	Transformational	Vector multiplication
DPROD	F90	Elemental	Numeric
EOSHIFT	Yes	Transformational	Array manipulation
EPSILON	F90	Inquiry	Numeric inquiry
EXP	F77	Elemental	Numeric
EXPONENT	F90	Elemental	Floating point man.
FLOOR	F90	Elemental	Numeric
FRACTION	F90	Elemental	Floating point man.
HUGE	F90	Inquiry	Numeric inquiry
IACHAR	F90	Elemental	Transfer
IAND	Mil. Std.	Elemental	Bit computation
IBCLR	Mil. Std.	Elemental	Bit computation
IBITS	F90	Elemental	Transfer
IBSET	Mil. Std.	Elemental	Bit computation
ICHAR	F90	Elemental	Transfer
IEOR	Mil. Std.	Elemental	Bit computation
INDEX	F90	Elemental	Character
INT	F90	Elemental	Transfer
IOR	Mil. Std.	Elemental	Bit computation
ISHFT	Mil. Std.	Elemental	Bit computation
ISHFTC	Mil. Std.	Elemental	Bit computation
KIND	Yes	Inquiry	Kind
LBOUND	Yes	Inquiry	Array inquiry
LEN	F90	Inquiry	Char inquiry
LEN_TRIM	F90	Elemental	Character
LGE	F77	Elemental	Character
LGT	F77	Elemental	Character
LLE	F77	Elemental	Character
LLT	F77	Elemental	Character
LOG	F77	Elemental	Numeric
LOG10	F77	Elemental	Numeric
LOGICAL	F90	Elemental	Transfer
MATMUL	Yes	Transformational	Array multiplication
MAX	F77	Elemental	Numeric
MAXEXPONENT	F90	Inquiry	Numeric inquiry
MAXLOC	restricted	Transformational	Array location
MAXVAL	Yes	Transformational	Array reduction
MERGE	Yes	Elemental	Array construction
MIN	F77	Elemental	Numeric
MINEXPONENT	F90	Inquiry	Numeric inquiry

Intrinsic	Available	Class	Type
MINLOC	Restricted	Transformational	Array location
MINVAL	Yes	Transformational	Array reduction
MOD	F77	Elemental	Numeric
MODULO	F90	Elemental	Numeric
MVBITS	F90	Subroutine	Subroutine
NEAREST	F90	Elemental	Floating point man.
NINT	F77	Elemental	Transfer
NOT	Mil. Std.	Elemental	Bit Computation
NULL	No	Transformational	Pointer association
PACK	serial	Transformational	Construction
PRECISION	F90	Inquiry	Numeric inquiry
PRESENT	Yes	Inquiry	Argument
PRODUCT	Yes	Transformational	Array reduction
RADIX	F90	Inquiry	Numeric inquiry
RANDOM_NUMBER	Yes	Subroutine	Subroutine
RANDOM_SEED	Yes	Subroutine	Subroutine
RANGE	F90	Inquiry	Numeric inquiry
REAL	F77	Elemental	Transfer
REPEAT	F90	Transformational	Character
RESHAPE	serial	Transformational	Array reshape
RRSPACING	F90	Elemental	Floating point man.
SCALE	F90	Elemental	Floating point man.
SCAN	F90	Elemental	Character
SELECTED_INT_KIND	Yes	Transformational	Kind
SELECTED_REAL_KIND	Yes	Transformational	Kind
SET_EXPONENT	F90	Elemental	Floating point man.
SHAPE	Yes	Inquiry	Array inquiry
SIGN	F77	Elemental	Numeric
SIN	F77	Elemental	Numeric
SINH	F77	Elemental	Numeric
SIZE	Yes	Inquiry	Array inquiry
SPACING	F90	Elemental	Floating point man.
SPREAD	Yes	Transformational	Array construction
SQRT	F77	Elemental	Numeric
SUM	Yes	Transformational	Array reduction
SYSTEM_CLOCK	Yes	Subroutine	Subroutine
TAN	F77	Elemental	Numeric
TANH	F77	Elemental	Numeric
TINY	F90	Inquiry	Numeric inquiry
TRANSFER	F90	Transformational	Transfer
TRANSPOSE	Yes	Transformational	Array manipulation
TRIM	F90	Transformational	Character
UBOUND	Yes	Inquiry	Array inquiry
UNPACK	serial	Transformational	Array construction
VERIFY	F90	Elemental	Character

Intrinsic	Available	Class
ILEN	No	Elemental
NUMBER_OF_PROCESSORS	Yes	System inquiry function
PROCESSORS_SHAPE	Yes	System inquiry function
ACTIVE_NUM_PROCS	Yes	System inquiry function
ACTIVE_PROCS_SHAPE	Yes	System inquiry function
TRANSPOSE	No	Transformational

Table 4: HPF Intrinsic Procedures.

Routine	Available	Class
ALL_PREFIX	No	Transformational
ALL_SCATTER	Yes	Transformational
ALL_SUFFIX	No	Transformational
COPY_PREFIX	No	Transformational
COPY_SCATTER	Yes	Transformational
COPY_SUFFIX	No	Transformational
COUNT_PREFIX	No	Transformational
COUNT_SCATTER	Yes	Transformational
COUNT_SUFFIX	No	Transformational
ANY_PREFIX	No	Transformational
ANY_SCATTER	Yes	Transformational
ANY_SUFFIX	No	Transformational
GRADE_DOWN	No	Transformational
GRADE_UP	No	Transformational
HPF_ALIGNMENT	Yes	Mapping inquiry subroutine
HPF_DISTRIBUTION	Yes	Mapping inquiry subroutine
HPF_TEMPLATE	Yes	Mapping inquiry subroutine
IALL	Yes	Transformational
IALL_PREFIX	No	Transformational
IALL_SCATTER	Yes	Transformational
IALL_SUFFIX	No	Transformational
IANY	Yes	Transformational
IALL_PREFIX	No	Transformational
IALL_SCATTER	Yes	Transformational
IALL_SUFFIX	No	Transformational
IPARITY	Yes	Transformational
IPARITY_PREFIX	No	Transformational
IPARITY_SCATTER	Yes	Transformational
IPARITY_SUFFIX	No	Transformational
LEADZ	No	Transformational
MAXVAL_PREFIX	No	Transformational
MAXVAL_SCATTER	Yes	Transformational
MAXVAL_SUFFIX	No	Transformational
MINVAL_PREFIX	No	Transformational
MINVAL_SCATTER	Yes	Transformational
MINVAL_SUFFIX	No	Transformational

Table 5: Routines of HPF_LIBRARY.

Routine	Available	Class
PARITY	Yes	Transformational
PARITY_PREFIX	No	Transformational
PARITY_SCATTER	Yes	Transformational
PARITY_SUFFIX	No	Transformational
POPCNT	No	Elemental function
POPPAR	No	Elemental function
PRODUCT_PREFIX	No	Transformational
PRODUCT_SCATTER	Yes	Transformational
PRODUCT_SUFFIX	No	Transformational
SORT_DOWN	No	Transformational
SORT_UP	No	Transformational
SUM_PREFIX	No	Transformational
SUM_SCATTER	Yes	Transformational
SUM_SUFFIX	No	Transformational

Table 6: Routines of HPF_LIBRARY (contd.).

Routine	Available	Class
GLOBAL_ALIGNMENT	Yes	Inquiry
GLOBAL_DISTRIBUTION	Yes	Inquiry
GLOBAL_TEMPLATE	Yes	Inquiry
GLOBAL_SHAPE	Yes	Inquiry function
GLOBAL_SIZE	Yes	Inquiry function
ABSTRACT_TO_PHYSICAL	No	Subroutine
PHYSICAL_TO_ABSTRACT	No	Subroutine
LOCAL_TO_GLOBAL	No	Subroutine
GLOBAL_TO_LOCAL	No	Subroutine
MY_PROCESSOR	Yes	Pure function
LOCAL_BLKCNT	No	Pure function
LOCAL_LINDEX	No	Pure function
LOCAL_UINDEX	No	Pure function

Table 7: Routines of HPF_LOCAL_LIBRARY

Routine	Available	Class
F77_GLOBAL_ALIGNMENT	Yes	Inquiry
F77_GLOBAL_DISTRIBUTION	Yes	Inquiry
F77_GLOBAL_TEMPLATE	Yes	Inquiry
F77_GLOBAL_SHAPE	Yes	Inquiry function
F77_GLOBAL_SIZE	Yes	Inquiry function
F77_ABSTRACT_TO_PHYSICAL	No	Subroutine
F77_PHYSICAL_TO_ABSTRACT	No	Subroutine
F77_LOCAL_TO_GLOBAL	No	Subroutine
F77_GLOBAL_TO_LOCAL	No	Subroutine
F77_MY_PROCESSOR	Yes	Pure function
F77_LOCAL_BLKCNT	No	Pure function
F77_LOCAL_LINDEX	No	Pure function
F77_LOCAL_UINDEX	No	Pure function

Table 8: F77 callable routine of Local Library.

Routine	Available	Class
HPF_TASK_INIT	Yes	Subroutine
HPF_TASK_EXIT	Yes	Subroutine
HPF_TASK_RANK	Yes	Subroutine
HPF_TASK_SIZE	Yes	Subroutine
HPF_SEND	Yes	Subroutine
HPF_RECV	Yes	Subroutine
HPF_SEND_INIT	Yes	Subroutine
HPF_RECV_INIT	Yes	Subroutine
HPF_TASK_COMM	Yes	Subroutine

Table 9: Routines of HPF_TASK_LIBRARY.

References

[ABM+92] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smit, and L. Wagener, J. *Fortran 90 Handbook*. Intertext-McGraw Hill, New York, NY, 1992.

[Hig94] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 1.1, Department of Computer Science, Rice University, November 1994.

[Hig97] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 2.0, Department of Computer Science, Rice University, January 1997.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[MEKN96] J. Moreira, K. Eswar, R. Konuru, and V. Naik. Supporting Dynamic Data and Processor Repartitioning for Irregular Applications. IBM Research Report RC20426, IBM, April 1996.

[Thi91] Thinking Machines Corporation. CM Fortran Programming Guide, Version 1.0. Manual, TMC, January 1991.

Index