Contents

ADAPTOR
Language Reference Manual
Version 1.0 (June 1993)

T. Brandes

Internal Report No. Adaptor 3

July 3, 1993

High Performance Computing Center
German National Research Institute for Computer Science
P. O. Box 1316
D-5205 Sankt Augustin 1
Federal Republic of Germany
Tel.: +49 (0)2241 / 14-2492
E-mail: Thomas.Brandes@gmd.de

ADAPTOR
Language Reference Manual
Version 1.0 (June 1993)

T. Brandes

German National Research Center for Computer Science,
P.O. Box 1316, D-5205 Sankt Augustin 1, FRG

Abstract

ADAPTOR (Automatic DAta Parallelism TranslatOR) is a tool that transforms data parallel programs written in Fortran with array extensions, parallel loops, and layout directives to parallel programs with explicit message passing. The input language of Adaptor is a mix of Fortran 90, Connection Machine Fortran and High Performance Fortran though not all features of these languages are supported.

The generated message passing programs will run on different multiprocessor systems with distributed memory, but also on shared or virtual shared memory architectures.

In this paper the source language for Adaptor is described. It will be shown which language extensions are supported by the tool to generate efficient parallel 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.

The source language of Adaptor is Fortran 77 with many features of Fortran 90, Connection Machine Fortran (CMF) [Thi91] and High Performance Fortran (HPF) [Hig93].

The following extensions of Fortran 90 can be used within Adaptor:

- array expressions and array assignments,
- intrinsic functions for arrays,
- dynamic arrays,
- 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.

As the tool has been designed originally to run CM Fortran programs on MIMD architectures, the source language is strongly related to CM Fortran. The following CMF features are supported:

- layout directives of CM Fortran,
- parallel random numbers,
- timing functions,
- global send and global get (scatter/gather operations),
- `FORALL` statement.

The following HPF features are supported:

- distribution directives of HPF,
- `FORALL` statement,
- `PURE` subprograms,
- some new intrinsic functions.

Some other features of Adaptor can also be used though these features are more intended for internal use:

- independent loops with local access
- overlapping of arrays

Adaptor does not support all features of Fortran 90 and HPF. The most important restrictions are:

- no modules,
- no pointer,

- no array-valued functions,
- no assumed-shaped arrays,
- only block distribution along one dimension,
- no explicit alignment (of course there is an implicit alignment for arrays that are declared in the same way)

Adaptor is not a compiler but a source to source transformation that generates Fortran 77 host and node programs with message passing. The new generated source code has to be compiled by the compiler of the parallel machine. Therefore the generated code will contain only features that are supported by this compiler.

In this way all extensions and directives supported by Adaptor that are not part of Fortran 90 are translated to Fortran 77 with corresponding library calls. By setting an option it is also possible to translate the code to a Fortran 90 program with message passing.

2 Basic Partitioning Features

The best use of the parallel hardware will be given if the arrays of the data parallel program are distributed among the nodes and the node processors operate simultaneously on the their local parts of the distributed arrays. Communication of non-local data should be kept as minimal as possible.

The partitioning of a data parallel program consists of the following three tasks:

- distribution of arrays,
- distribution of parallel loop iterations,
- generation of communication for distributed data movements.

The translation strategy of Adaptor reflects the idea that the whole code is executed on every process, but array operations or parallel loops are distributed corresponding to the data distribution. I/O statements are executed only by the host process.

2.1 Distribution of Arrays

A layout directive enables the user to specify the dimension of an array which should be distributed. In the current version only block distribution along one dimension is supported.

```
      INTEGER a(100), b(100,n), c(100,n), x(10)
!HPF$ DISTRIBUTE a(BLOCK)
!HPF$ DISTRIBUTE b(*,BLOCK)
!HPF$ DISTRIBUTE c(BLOCK,*)
!HPF$ DISTRIBUTE x(*)          ! replicated on all processors
```

The following example shows how arrays with 17 elements in the last dimension are distributed onto 5 processors:

```
     REAL a(17), b(10,17)
!HPF$ DISTRIBUTE a(BLOCK)
!HPF$ DISTRIBUTE b(*,BLOCK)

    --------------------------------------------------------------
    |  a(1:3)  |  a(4:6)  |  a(7:10)  |  a(11:13)  |  a(14:17)  |
    --------------------------------------------------------------
    | b(:,1:3) | b(:,4:6) | b(:,7:10) | b(:,11:13) | b(:,14:17) |
    --------------------------------------------------------------
       P(1)       P(2)       P(3)         P(4)         P(5)
```

2.2 The Local Independent Do Loop

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.

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

2.2.1 Home of Iterations

The first array variable on the left hand side of an assignment decides how the iterations are distributed.

```
!HPF$ INDEPENDENT, LOCAL_ACCESS
   DO i = 1, n
      x(i) = ....
      ....
   END DO
```

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

2.2.2 Scalar Variables

Scalar variables can be defined within a local independent do loop. The semantic is that every processor uses its own incarnation of the variable. But if a replicated variable is defined within the loop, the value will be undefined after all iterations.

```
!HPF$ INDEPENDENT, LOCAL_ACCESS
   DO i = 1, n
      s    = ...
      x(i) = x(i) * s
   END DO
```

2.2.3 Simple Reductions

In many cases it is necessary to make a reduction in the parallel loop. This is supported with the reduce statement. This statement can only appear in a local independent do loop.

```
   REDUCE (function, variable, expression)
```

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, LOCAL_ACCESS
      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 avaialable:

- `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

2.2.4 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, LOCAL_ACCESS
      DO i2 = 1, n
         DO i1 = 1, n
            REDUCE (MINVAL, min, b(i1,i2), imin1, i1, imin2, i2)
         END DO
      END DO
```

2.3 Moving Distributed Data

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

There are many other possibilities for moving distributed data efficiently. This will be discussed in section 4.

2.4 Scalar Code

Scalar assignments to a variable are executed by the processor that owns the defined variable. For an assignment to a scalar variable the value of the scalar expression has to be evaluated by all processors.

2.4.1 Scalar Expressions

A scalar expression will be evaluated by all processors, if the expression is used

- to evaluate the condition of control flow statements (if conditions, loop conditions, etc.),
- to assign a value to a replicated scalar variable.

These scalar expressions maz contain variable accesses to distributed and host arrays. If a value of a scalar data is needed by every processor, this value will be broadcast by the owner processor of this data. If the scalar data is a replicated variable, no broadcast will be necessary as every processor performs the updates of a replicated variable on its own.

```
      REAL a(n)
!HPF$ DISTRIBUTE a(BLOCK)
      REAL g
      ...
      g = a(51)    ! one node broadcasts the value
```

2.4.2 Scalar Assignments

A scalar assignment updates a scalar data. If this data is replicated, every processor will perform its own update. If the data is distributed, only the owner of this data will make the necessary update.

```
      REAL a(n)
!HPF$ DISTRIBUTE a(BLOCK)
      REAL g

      a(5) = g        ! only owner of a(5) makes the update
```

If the expression in the assignment contains accesses to elements of arrays that are not located on the processor, communication will become necessary.

```
      a(5) = a(5) + 1 ! update without communication
      a(5) = a(8)     ! one node sends a value to another node
```

It should be observed that scalar assignments with implicit communication are very inefficient as every single value is sent in one communication and the start-up time is needed for every value.

2.5 Example Program

The following example program shows a typical data parallel program using the discussed features.

```
      PROGRAM prime
      INTEGER n, s, k
      PARAMETER (n=10000)
      LOGICAL*1 a(n)
!HPF$ DISTRIBUTE a(BLOCK)
      a(1) = .FALSE.
!HPF$ INDEPENDENT, LOCAL_ACCESS
      DO i = 2, n
        a(i) = .TRUE.
      END DO
      k = 2
      DO WHILE (k*k <= n)
!HPF$ INDEPENDENT, LOCAL_ACCESS
         DO i = k*k, n, k
c          all multiples of k are no primes
           a(i) = .FALSE.
         END DO
         k = k + 1
         DO WHILE (.NOT. a(k))
           k = k + 1
         END DO
      END DO
      s = 0
```

```
!HPF$ INDEPENDENT, LOCAL_ACCESS
      DO i = 1, n
        REDUCE (COUNT, s, a(i))
      END DO
      PRINT *, 'There are ',s,' primes until ', n
      END
```

3 Layout of Arrays

In this section it is described how the variables of the data parallel program can be distributed among the host and nodes for the parallel execution. As the distribution of variables decides how many communication will be generated, it is an issue of program optimization but not an issue of correctness.

The distribution or layout of variables is specified by compiler directives very similar to those used in CM Fortran or High Performance Fortran.

Scalar variables are always replicated. Arrays can be

- *replicated* (every processor has an own incarnation),
- *distributed* (distributed among all available node processors),
- *host arrays* (owner is the host process).

If an array is replicated or a host array, it will be called sequential. Data of a sequential array is stored in the sequential order assumed by Fortran 77 semantic.

3.1 Explicit Distributions

3.1.1 Replicated Arrays

Usually array variables are distributed among the node processors to avoid high memory overhead. In some cases, e.g. for small data arrays, it might be useful to replicate them.

There are some possibilities to specify that an array is replicated. Directives from CMF or HPF can be used.

```
      INTEGER a(100), b(100,n)
CMF$  LAYOUT a(:REPLICATED), b(:REPLICATED)

      INTEGER a(100), b(100,n)
CMF$  LAYOUT a(:SERIAL), b(:SERIAL,:SERIAL)
```

The SERIAL attribute must be given for every dimension. The REPLICATED attribute is only required once.

```
      INTEGER a(100), b(100,n)
!HPF$ DISTRIBUTE a (*)
!HPF$ DISTRIBUTE b (*,*)
```

If compared to CM Fortran, replicated arrays are not front end arrays. Every processors has its own incarnation but the transformation guarantees that all instances have always the same values at least at the synchronization points.

If the data parallel program has only replicated arrays, every processor will execute all statements of the input program. There is only one exception: only one process executes I/O operations and calls external subroutines.

3.1.2 Distributed Arrays

In the current version of the Adaptor tool array variables can only be distributed along one dimension. For the distribution the block distribution is used. The next version will also support cyclic distributions and two-dimensional distributions.

The following CMF layout or HPF distribution directives can be used to specify the dimension of an array that is intended to be distributed:

```
      INTEGER a(100), b(100,n), c(100,n)
CMF$  LAYOUT a(:NEWS), b(:SERIAL,:NEWS), c(:NEWS,:SERIAL)

      INTEGER a(100), b(100,n), c(100,n)
!HPF$ DISTRIBUTE a(BLOCK)
!HPF$ DISTRIBUTE b(*,BLOCK)
!HPF$ DISTRIBUTE c(BLOCK,*)
```

3.1.3 Host Arrays

Host arrays have only one incarnation in the host program. The following reasons tescribe the intention of a host array:

- an array is used in an external subroutine (e.g. for X-Windows),
- an array is used for input or output (input and output of distributed arrays is not supported until now),
- an array that would be useful to be replicated does not fit in the memory of the nodes (it can be transferred in blocks from host to nodes, using the virtual memory of the host operating system).

The following layout directive declares an array to be a host array:

```
      INTEGER ha(100), hb(100,n)
CMF$  LAYOUT a(:HOST), b(:HOST)
```

It should be mentioned that the HOST directive has to be specified only once and not for every dimension.

3.2 Indirect Distributions

Arrays can be aligned with templates. The distribution of the template implies the distribution of the aligned arrays.

In the current version the alignment is not realized in the sense of CM Fortran or High Performance Fortran. The alignment only decides which dimension will be distributed. In which way the dimension is distributed depends only on the size of the array.

The following examples 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)

      REAL*8  uo (5,nqpx,nqpx,nkpx)
      REAL*8  u  (5,nqpx,nqpx,nkpx)
      REAL*8  up (5,nqpx,nqpx,nkpx)
      REAL*8  fu (5,nqpx,nqpx,nkpx)
      REAL*8  gu (5,nqpx,nqpx,nkpx)
      REAL*8  hu (5,nqpx,nqpx,nkpx)
      REAL*8  ku (5,nqpx,nqpx,nkpx)
      REAL*8  tmp  (nqpx,nqpx,nkpx)
      REAL*8  tmp1 (nqpx,nqpx,nkpx)

!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
```

If the distribution directive of the template is now changed to

```
!HPF$  DISTRIBUTE t ( *, BLOCK, *)
```

all other arrays will be distributed along the second last dimension.

3.3 Common Blocks and Sequence Association

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

3.3.1 Distributed Arrays in Common Blocks

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).

3.3.2 Sequence Association

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. This is also a kind of indirect distribution.

3.4 Default Distributions

If an array has not an explicit or indirect distribution, a default distribution will be created for this array. At the moment the user can choose between three possibilities:

- Every array is replicated by default.
- Every array is distributed in the last dimension by default (this does not apply for character arrays).

- Every array is distributed in the last dimension by default if it is used in an array statement. The same rules as used in CM Fortran will be applied.

In the following example no distribution has been specified for the arrays a and b, respectively. If the rules of CM Fortran apply, array `a` is distributed and array `b` becomes a replicated array.

```
PROGRAM simple
REAL a(100), b(100)
a = b(1)
END
```

4 Data Movement with Array Sections

In the next two sections it is described which kind of data movements are supported by Adaptor in the sense that they are realized efficiently.

In this section data movements on primitive array assignments are discussed.

4.1 Array Sections and Constructors

An array section is a subset of the elements in an array. An array section can be referred to by using section selectors in place of a subscript specifier in a reference to an array. Section selectors and scalar subscripts can be mixed in references to multi-dimensional arrays.

A section selector takes one of the following forms:

- Indexed section selector `[begin]:[end][:stride]`
- Vector-valued section selector that is an array expression that must represent a 1-dimensional integer array whose elements index the selected positions in the primary array

```
DIMENSION a(9), b(4,9), m(3)
...
a(1:5)
b(m,5)
b(1:3,:)
```

In some cases contiguous array sections are required. A section will be contiguous, if all elements of the section are lying consecutively in memory. Therefore, a contiguous array section is one in which the sections selectors

- are indexed with a stride of 1,
- appear to the left of any scalar subscripts,
- include every element of the dimensions except that the rightmost selector (before any scalar subscripts) can truncate the dimension

```
DIMENSION a(4,9)

a(:,1:5)    ! contiguous
a(1:3,2)    ! contiguous
a(:,1:5:2)  ! noncontiguous
a(3,1:5)    ! noncontiguous
a(1:3,:)    ! noncontiguous
```

An array value can be formed with an array constructor. In the current version an array constructor can only be

- a range specifier `'['begin:end[:stride]']'`.

An array constructor is always 1-dimensional.

4.2 Primitive Array Assignments

A primitive array assignment is an assignment in which the left-hand side of the statement is an array or array section. The expression of the right-hand side of the statement is a primary (array, array section, array constructor, scalar value). Both arrays must be conformable.

Two arrays are conformable if they have the same shape. A scalar value is conformable to any array.

```
a = b
a(1:n) = b(2:n+1)
a(1:n) = [2:n+1]
a(1:n) = a(5)
```

The expression of a primitive array assignment will be aligned with the array or array section on the left hand side, if the assignment requires no communication.

```
      REAL a(n), ha(n), ra(n)
      REAL g
!HPF$ DISTRIBUTED a  (BLOCK)  ! a is distributed
CMF$  LAYOUT ha (:HOST)       ! ha is host array
```

```
!HPF$ DISTRIBUTED ra (*)      ! ra is replicated

      a(1:n) = ra(1:n)        ! aligned
      ra(1:n) = a(1:n)        ! not aligned

      ra(2:n) = ra(1:n-1)     ! aligned
      a(2:n)  = a(1:n-1)      ! not aligned
      ha(2:n) = ha(1:n-1)     ! aligned
```

Due to different possibilities of the layout of the array variables, different kinds of communications might be generated. At the moment not all primitive assignments are allowed. The following rules should be observed, if the statement needs communication:

- No strides are allowed in sections of distributed arrays
- The sections of a host array or a replicated array must be contiguous
- The left and the right hand side must have the same type.

4.3 Moving Sections of Distributed Arrays

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.

In the current version it is impossible to use strides in any data movement that requires communication.

```
      REAL a(n,n), a1(n,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: A, A1
      REAL g

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

      a(1,1:n:2) = a(2,1:n:2)      ! no communication
      a(1:n-1:2,:) = a(2:n:2,:)    ! no communication
      a(1:n-1,1:n-1) = a(2:n,2:n)  ! communication
      a(1:n,1) = a(n:1:-1,2)       ! requires communication; but stride
                                   ! is not 1, so it is not allowed
```

4.4 Replication of a Section of a Distributed Array

In many cases it is necessary to replicate a section of a distributed array. This is done by assigning this section to a replicated array.

The replicated array section on the left hand side of the assignment must be contiguous.

```
      REAL a(n,n), ra(n,n), ra1(n)
!HPF$ DISTRIBUTE (*,BLOCK) :: a
!HPF$ DISTRIBUTE (*)       :: ra1
!HPF$ DISTRIBUTE (*,*)     :: ra
      INTEGER k
      REAL g

      ra = a          ! replication of a whole distributed array

      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

      ra(1:n,1) = a(k,1:n)  ! allowed, as ra(1:n,1) is contiguous
      ra(1,1:n) = a(1:n,1)  ! sorry, ra(1,1:n) is not contiguous
```

4.5 Moving Host Arrays

A primitive assignment with host arrays can be used to move data from host arrays to distributed arrays on the nodes and vice versa.

The section of the host array must always be contiguous.

```
      REAL a(n,n), ha(n,n), ra(n)
!HPF$ DISTRIBUTE (*,BLOCK) :: a
!HPF$ DISTRIBUTE (*)       :: ra
CMF$  LAYOUT ha (:HOST)
      REAL g

      ha = a    ! distributed array is moved to host, efficient

      ra = ha   ! not allowed in the current version
      ha = ra   ! allowed, only executed by the host

      ha(1:n,1) = a(1,1:n)  ! allowed, as ha(1:n,1) is contiguous
      ha(1,1:n) = a(1:n,1)  ! sorry, ha(1,1:n) is not contiguous
```

5 Data Movement with Intrinsics

Array intrinsic functions are another possibility to exchange data between processors. In the current version only some functions are supported.

5.1 Circular Shifting

Adaptor supports circular shifting only for whole arrays. Source and target array must have the same shape, both arrays must be sequential or distributed.

```
      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 not supported.

5.2 Transpose

Adaptor supports the transposing of whole two-dimensional arrays. Source and target array must have the same shape, both arrays must be sequential or distributed.

```
      REAL a(m,n), b(n,m)
!HPF$ DISTRIBUTE (*,BLOCK) :: a, b
      ...
      a = TRANSPOSE (b)
```

It should be mentioned that in some situations an implicit transpose is possible.

```
      REAL a(m,n), b(m,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: a
!HPF$ DISTRIBUTE (BLOCK,*) :: b
      ...
      a = b     ! implicit transpose due to index switching
```

5.3 Indirect Addressing

In most cases it is necessary that the indexes of array sections are replicated. The current version of Adaptor supports only one exception: the indexing with whole distributed integer arrays.

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

b(p) = a                   ! global send
CALL GLOBAL_SEND (b,p,a)   ! the same operation

a = b(p)                   ! global get
CALL GLOBAL_GET (a,b,p)    ! the same operation
```

5.3.1 Gathering of Data

The following intrinsic subroutine is used to gather data from the array B:

```
CALL GLOBAL_GET (A, B, P1, ..., PN, MASK)
```

The `MASK` parameter is optional.

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 global get operation will be described by the following loop nesting:

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

5.3.2 Scattering of Data

The following intrinsic subroutine is used to scatter data from an array A to an array B:

```
CALL GLOBAL_SEND (B, P1, ..., Pn, A, MASK, red_function)
```

The `MASK` parameter and the reduction function parameter are optional. If the reduction function is missing, the data of A will be copied into the array B, otherwise the reduction function will be applied to the old element of B together with the new one.

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 global send 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
```

The semantic of the scatter intrinsic subroutine `global_send` is very similar to the array combining scatter functions `XXX_scatter` of HPF [Hig93]. The main difference is given by the fact that

- here a subroutine is used and not an intrinsic function,
- and the kind of reduction is specified with a parameter and not with a prefix in the name of the operation.

6 Overlapping of Arrays

In local independent loops only local data can be used or defined. In some cases it is very useful to define arrays that have an overlap area. By this way local access to neighbored elements is possible.

6.1 Definition of an Overlap Area

```
REAL <array_name> ( [low1:]up1 [ '['left_overlap1:right_overlap1']' ] ,
                    [low2:]up2 [ '['left_overlap2:right_overlap2']' ] ,
                    ...
                    [lowk:]upk [ '['left_overlapk:right_overlapk']' ] )
```

The values of the overlap specifications must be non-negative integer constants. The values define the size of the overlapping area. If no overlapping is specified the values will be assumed to be zero.

After the source to source translation has been made the size of the arrays will correspondingly be extended to the overlap size.

```
     REAL a(1:n[1:1],n[2:2])
c    becomes
     REAL a(0:n+1,-1:n+2)
```

6.2 Use of Arrays with Overlapping

The usage of arrays with overlap areas is limited to some cases. Arrays with an overlap area can only be defined in a single assignment. In this case the local values will be copied and the overlap area will be exchanged between the processes.

```
      REAL a   (n, n)
      REAL ova (n[1:1],n[2:2])
!HPF$ DISTRIBUTE (BLOCK) :: a, ova
      ...
      ova = a
```

Arrays with an overlap area can only be used in a local context. This means that they can only be used within statements that will not require any communication.

```
   REAL a   (n, n)
   REAL ova (n[1:1],n[2:2])

   ova = a   ! ova is copy of a with correct boundaries
!HPF$ INDEPENDENT, LOCAL_ACCESS
   DO j = 2, n-1
!HPF$ INDEPENDENT, LOCAL_ACCESS
      DO i = 2, n-1
         a(i,j) = ova(i-1,j) + ova(i+1,j) + ova(i,j-1) + ova(i,j+1)
      END DO
   END DO
```

If ova had not an overlap area, some of its values would not be local in the loop.

7 Array Statements

Adaptor supports array statements. Attention should be paid if communication is necessary or not.

7.1 Local Array Assignments

An array assignment will be a local assignment if it needs no communication between the different processors.

```
      REAL a(n), ha(n), ra(n)
      REAL b(m,n)
      REAL g
CMF$  LAYOUT a(:NEWS)         ! a  is distributed
CMF$  LAYOUT b(:NEWS,:SERIAL) ! b  is distributed
CMF$  LAYOUT ha(:HOST)        ! ha is a host array
CMF$  LAYOUT ra(:SERIAL)      ! ra is a replicated array

      ha = ra       ! only executed by host
      ha = ha + 1   ! local on host

      ra = ra + 1   ! local on every node (no data parallelism)
      a  = ra       ! local

c     The following statements are executed on the nodes and take
c     full advantage of the data parallelism
```

```
      a   = a * 2
      a   = a + b(:,4)
      a   = a + g
      b   = SPREAD (ra, 1, m)
```

Adaptor recognizes this kind of local operation and does not generate communication. As the operation is distributed corresponding to the distribution of the data among the processors, these operations result in good speed-ups.

7.2 Array Assignments with Communication

In many cases array assignments need communication. In this case Adaptor splits up the assignment in primitive array assignments with communication and local array assignments.

```
      REAL a(n), b(n), c(n)
!HPF$ DISTRIBUTE (BLOCK) :: a, b, c

      a(1:k) = b(1:k) - c(k+1:n-k)
```

This array assignment is split into two statements, one data movement and a local array assignment.

```
      a(1:k) = c(k+1:n-k)
      a(1:k) = b(1:k) - a(1:k)
```

For this example no temporary array is required. This might be unavoidable for other array assignments.

```
      a(2:n-1) = ( b(1:n-2) + b(3:n) ) * 0.5
```

This array assignment requires at least one temporary array.

```
      tmp_a(2:n-1) = b(1:n-2)
      a(2:n-1) = b(3:n)
      a(2:n-1) = (tmp_a(2:n-1) + a(2:n-1) ) * 0.5
```

The splitting of non-local assignments is a non-trivial task as not too many or too big temporaries should be created. As this task has not been completely solved, it might be possible that Adaptor has done this job not well. It is also possible that Adaptor will fail to translate a complex statement with many array operations.

```
      REAL a(n), ha(n), ra(n)
      REAL g
CMF$  LAYOUT a(:NEWS)         ! a  is distributed
CMF$  LAYOUT ha(:HOST)        ! ha is a host array
CMF$  LAYOUT ra(:SERIAL)      ! ra is a replicated array

c     this is not possible
      a(51:71) = ra(49:69) - ha(25:45) * SUM (a(17:88))

c     but
      g = SUM(a(17:88))
      a(51:71) = ra(49:69) - ha(25:45) * g
```

8 The WHERE Statement

The block `WHERE` statement is used to assign each element in an array assignment conditionally.

The body of the block in the where statement must contain only array assignments. It is not allowed to transfer control in a where block or end a do loop in a where block. Where blocks cannot be nested.

```
      WHERE (b .GT. 0.0)
           a = s
        ELSEWHERE
           a = 0.0
      ENDWHERE
```

Occasionally, some further restrictions are given for the current version of Adaptor:

- Variables within the mask of the where block can only be updated in the last statement of the body,
- the mask should be aligned with every array or array section of the left hand side of the assignments

```
      REAL a(n,n), b(n,n), c(n)
!HPF$ DISTRIBUTE (*,BLOCK) :: a, b
!HPF$ DISTRIBUTE (BLOCK)   :: c

c     allowed, no communication necessary
      WHERE (b(1,:) .GT. 0)
        b(2,:) = b(3,:)
      END WHERE
```

```
c     allowed, no communication required for the mask
      WHERE (b(1,:) .GT. 0)
        b(2,:) = CSHIFT (c,1,1)
      END WHERE

c     not allowed as update of B not only in the last statement
      WHERE (b(1,:) .GT. 0)
        b(2,:) = b(3,:)
       ELSEWHERE
        b(2,:) = b(4,:) - 1.0
      END WHERE

c     not allowed as mask is not aligned with lhs of stmt
      WHERE (b(:,1) .GT. 0)
        b(:,2) = b(:,3)
      END WHERE
```

9 The FORALL Statement

A FORALL statement reveals to a collection of assignments to designated array elements [Lov92, WSG92, ALS91, Thi91]. As all these assignments can be done simultaneously, it is a natural way to express parallelism.

9.1 Syntax of 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.

```
     FORALL (v1=l1:u1:s1, ..., vn=ln:un:sn [,mask] )
    &     a(e1,...,em) = rhs
```

The parallelism of this assignment is given by the fact that the assignment can be executed in any order.

The following constraints have to be observed:

- The assignment must not cause any element of the lhs-array to be assigned a value more than once.
- The mask expression may depend on the subscript-names.
- Each of the subscript-names must appear within the subscript expression(s) on the left-hand-side.

```
      FORALL (i=1:n, j=1:n) h(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)
```

The following statement is illegal as the subscript-name j does not appear on the left hand side.

```
c     FORALL (i=1:n, j=1:n) h(i) = h(i) + 1.0 / REAL(i+j-1)
```

9.2 Local FORALL Statement

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

The FORALL statement will be local if it requires no communication. The following ones are all local.

```
      REAL h(n,n), a(n,n), b(n,n)
!HPF$ DISTRIBUTE (*,BLOCK) h, a, b
      ...
      FORALL (i=1:n, j=1:n) h(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)
```

Local FORALL statements will be translated directly to local independent loops.

At the moment Adaptor rejects local FORALL statements where the corresponding do loop might have data flow dependences.

```
      REAL a(n,n)
!HPF$ DISTRIBUTE (*,*)
      ...
      FORALL (i=1:n, j=2:n-1) a(i,j) = (a(i,j-1) + a(i,j+1)) * 0.5
```

9.3 FORALL Statements with Communication

In many cases a FORALL statement will 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
```

In contrary to the local independent loops, required communication for data is recognized and communication statements will be generated.

9.4 FORALL Statements with Array Statements

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

```
      REAL a(n)
!HPF$ DISTRIBUTE a(BLOCK)
      ...
      FORALL (i=1:n) h(1:i,i) = a(1:i,i)
```

Usually array statements will be no problem if they can be translated to forall or do loops. But Adaptor will impose some restrictions if intrinsic array functions are used, e.g. the following loop will not be translated.

```
c     not allowed for Adaptor are array assignments in FORALL
      FORALL (i=1:n, j=1:n) h(1:i,1:j) = TRANSPOSE (a(1:j,1:i))
```

10 Intrinsic Array Functions

The Fortran 77 intrinsic functions are extended to array operations and some intrinsics are new with array operations. Inquiry intrinsic functions cannot be used within Adaptor.

10.1 Elemental Intrinsic Functions

The elemental intrinsic functions are the same as supported for scalar processing. For array processing, at least one of the arguments is an array and the result is an array. All arguments and the result must be conformable.

The new elemental intrinsic function `MERGE` choosing one of two given values depending on a mask is not supported.

```
      REAL a(n), b(n)
      a = SQRT (b)
      a = SIN(b) + COS(b)
      a(1:n:2) = SIN(b(1:n:2))
```

10.2 Reductions

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

In contrary to a previous version, now `mask` and `dim` parameters in the reductions are also allowed.

```
      REAL a(n), ha(n), ra(n)
      REAL g
CMF$  LAYOUT a(:NEWS)         ! a  is distributed
CMF$  LAYOUT ha(:HOST)        ! ha is a host array
CMF$  LAYOUT ra(:SERIAL)      ! ra is a replicated array

      g = SUM(a)         ! global reduction to a replicated variable
      g = SUM(a(5:n))    ! global reduction to a replicated variable
```

10.3 Shifting, Transpose

Adaptor supports circular shifting and transposing only for whole arrays. These functions have been already discussed in section 5.1 and 5.2.

10.4 Transformational Intrinsic Functions

The transformational intrinsic functions produce results with a different shape than the arguments.

- spread, diagonal, pack, unpack, replicate, reshape
- matmul

The current version of Adaptor supports only the spread function. All other functions result in translation errors and cannot be used.

```
      REAL a(n), b(m,n), g
      ...
      b = SPREAD (SPREAD(g,1,n),1,m)
      b = SPREAD (a,1,m)
```

11 Dynamic Arrays

Adaptor 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).

It should be observed that for complex array statements Adaptor generates implicit allocations and deallocations for temporary arrays.

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

Dynamic arrays are always stored on the stack. Therefore the following rules have to be observed:

- an allocatable array must be currently allocated when it is passed as an actual argument to a subroutine,
- allocation is necessary before the first use,
- if a dynamic array is deallocated all arrays that have been allocated later will also be deallocated.

11.2 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)
```

12 Random Numbers

Similar to CM Fortran, Adaptor supports the generation of random numbers by using a parallelized random number generator.

- `cmf_randomize` initializes the random number generated with a seed value

- `cmf_random` generates random numbers for whole integer, real or double precision arrays

```
PROGRAM p
REAL x(100)
INTEGER na(100)
CALL CMF_RANDOMIZE(54)  ! 54 is seed for random function
CALL CMF_RANDOM (x)     ! assigns a random value to each element of x
CALL CMF_RANDOM (na,50) ! random values in range of 0 to 49
...
END
```

13 Timing

Adaptor generates very efficient code for well written data parallel programs. To prove this on your own, the following subroutines can be used to make time measurements:

- `cm_timer_clear`, `cm_timer_start`, `cm_timer_stop`, `cm_timer_print` give the possibility for machine independent timing.
- `walltime` gives access to the time since the program has started.

```
   PROGRAM p
   REAL s1, s2               ! replicated scalar variables
   ...
   CALL CM_TIMER_CLEAR (0)
   CALL CM_TIMER_START (0)
c  code to be measured
   ...
   CALL CM_TIMER_STOP (0)
   CALL CM_TIMER_PRINT (0)
   ...
   CALL WALLTIME (S1)
   CALL s(...)
   CALL WALLTIME (S2)
   print *, 'Execution of S needs ', S2-S1, ' seconds.'
```

14 I/O Operations

The central idea for I/O statements is that these statements are only executed by the host (Host-Node programming model) or by the first node (Only-Node programming model).

Every replicated variable that might be changed by an I/O statement is broadcast to all processors. I/O is not allowed for distributed variables.

When the necessity comes up to read in or to write distributed arrays, this should be done with the help of host arrays.

```
      REAL a(n,n), ha(n,n)
!HPF$ DISTRIBUTE a(*,BLOCK)
CMF$ LAYOUT ha(:HOST)

      READ *, ha       ! read in host array
      a = ha           ! distribute it to nodes
      ...              ! parallel operations on distributed array
      ha = a           ! collect distributed array to host
      PRINT *, ha      ! output of the host array
```

Here is a solution with replicated arrays.

```
      REAL a(n,n), ra(n)
!HPF$ DISTRIBUTE a(*,BLOCK)
!HPF$ DISTRIBUTE ra(*)

      DO i = 1, n
         READ *, ra    ! read in column i of a
         a(:,i) = ra   ! send it to process that owns a(:,i)
      END DO
      ...
      DO i = 1, n
         ra = a(:,i)   ! broadcast
         PRINT *, ra
      END DO
```

15 User Subprograms

Subprograms are absolutely necessary to structure bigger programs. A user subroutine is a subprogram that has been specified by the user and is available during the source-to-source translation. In contrary to an external subprogram, every processor runs into the subprogram.

15.1 Semantic of a User Subprogram

If the source program calls a user-defined subprogram, the generated host and node program will call this subprogram, too.

By this way it is possible that a subprogram can also have distributed arrays, parallel loops and implicit communication like the main program.

```
      PROGRAM p
      CALL s()        ! s will be called by every process
      CALL t()        ! t will be called by every process
      END

      SUBROUTINE s ()
      REAL a(n)
!HPF$  DISTRIBUTE a(block)
      ...
      END

      SUBROUTINE t ()
      REAL b(n), c(n)
!HPF$  DISTRIBUTE b(block)
!HPF$  DISTRIBUTE c(*)
      ...
      END
```

Common blocks can be used for global data without any problems. But if the common block contains distributed arrays all occurrences must have the same occurrence as described in 3.3.1.

15.2 Array Dummy Arguments of a Subprogram

The following general restrictions for dummy arguments are given:

- actual arrays are referenced by a pointer to the first element
- actual arrays cannot be array expressions
- assumed-shaped dummy arrays are not supported

```
  PROGRAM p
  REAL x(100)
  CALL s (x)         ! allowed subprogram call
  CALL s (x+1)       ! array expressions are not allowed
  ...
  END

  SUBROUTINE s (a)
C REAL a(:)          ! assumed-shaped arrays are not supported
  REAL a(100)        ! this is okay
  ...
  END
```

15.3 Distributed Dummy Arguments of a Subprogram

Distributed arrays can be actual or dummy parameters of a subprogram, too.

```
      SUBROUTINE doit (a, ha, ra, n, m)
      INTEGER n, m
      REAL a(n,m), ha(n,m), ra(n,m)
CMF$  LAYOUT ha(:HOST), ra(:SERIAL,:SERIAL)
!HPF$ DISTRIBUTE a(*,BLOCK)
      ...
      END
```

But the following restrictions have to be observed for distributed actual or dummy arrays:

- Actual and dummy array must have the same layout.
- If a distributed array is an actual array, the whole array will have to be the actual parameter.
- Assumed-sized dummy arrays cannot be distributed arrays.
- Arrays with overlap area cannot be actual or dummy arrays.

```
      PROGRAM P
      INTEGER n, m
      REAL a(n,m), ha(n,m), ra(n,m)
CMF$  LAYOUT ha(:HOST), ra(:SERIAL,:SERIAL)
!HPF$ DISTRIBUTE a(*,BLOCK)
      CALL doit (a, ha, ra, n, m)        ! absolutely okay
      CALL doit (a, ha, ra(2,4), n, m)   ! okay, ra replicated
      CALL doit (a(2,4), ha, ra, n, m)   ! wrong, a distributed
      CALL doit (a, ra, ha, n, m)        ! wrong, ha/ra wrong layout
      END
```

16 Pure Subprograms

In contrary to the user subprograms a pure subprogram can be called independently by different processors. It is assumed that a pure subprogram has only access to its own local data and does not have any side effects. No communication will be generated.

16.1 Syntax and Constraints

Pure subroutines 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 subroutine should not contain any distributed array. The actual arguments of a pure subprogram can be values from replicated data, host data or distributed data. But it must be guaranteed that one process owns all data of the arguments.

New local data within a subroutine will have an own incarnation on every processor. Therefore the SAVE statement is not allowed.

A pure subroutine must not have any I/O-Operations.

16.2 Use of Pure Subprograms

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

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

      REAL a(n,m), ra(n,m)
      INTEGER n, m
CMF$  LAYOUT ra(:SERIAL,:SERIAL)
      FORALL (i=1:n,j=1:m)
         a(i,j) = f(a(i,j), ra(i,j))
      END FORALL

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

      PROGRAM p
```

```
      REAL a(n)
!HPF$ DISTRIBUTE a(BLOCK)
      ...
!HPF$ INDEPENDENT, LOCAL_ACCESS
      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.

16.3 Realization of Pure Subprograms

The difference between a pure subroutine and a user subroutine for the source-to-source translation is that

- no temporary variables have to be created,
- and no communication must be generated.

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.

17 External Subprograms

External subroutines are called only by the host program. Therefore the use of external subroutines is very similar to the idea of I/O-statements.

A subroutine must not have a distributed array as an actual parameter. If one actual parameter is a replicated variable that could have been modified, the replicated values on the nodes will be updated by a broadcast.

It should be observed that a modification of common variables in an external subroutine has only an effect for the host values.

The call of external functions is not supported in the current version.

18 Example Program: Jacobi Iteration

18.1 Using Array Syntax

```
      PROGRAM LAPLACE
      REAL F(:,:), DF(:,:)
      LOGICAL CMASK (:,:)
      INTEGER MAXX, MAXY
      REAL FMAX
      INTEGER ITER
c
c     read in sizes
c
      PRINT *,'MAXX = (e.g. 64) '
      READ *,MAXX
      PRINT *,'MAXY = (e.g. 64) '
      READ *,MAXY
      ALLOCATE (F(MAXX,MAXY),DF(MAXX,MAXY),CMASK(MAXX,MAXY))
C
      CMASK = .FALSE.
      CMASK (2:MAXX-1,2:MAXY-1) = .TRUE.
      F = 2.
      F(:,MAXY) = 1.
      WHERE (CMASK)
          F = 0.0
      ENDWHERE
      ITER = 0
      FMAX = 1
      DO WHILE (FMAX .gt. 0.001)
         ITER = ITER + 1
         DF = (CSHIFT (F,1,1) + CSHIFT (F,1,-1) +
     &         CSHIFT (F,2,1) + CSHIFT (F,2,-1) ) * 0.25 - F
         WHERE (CMASK)
            F  = F + DF
           ELSEWHERE
            DF = 0.0
         ENDWHERE
         DF = ABS(DF)
         FMAX = MAXVAL (DF)
         PRINT *,'Iteration ',ITER,'  Max = ',FMAX
      END DO
      PRINT *, ITER, ' iterations needed'
      DEALLOCATE (CMASK, DF, F)
      END
```

18.2 Using FORALL Statements

The following solution uses `FORALL` statements.

```
      PROGRAM LAPLACE
      REAL F(:,:), DF(:,:)
      INTEGER MAXX, MAXY
```

```
      REAL FMAX
      INTEGER ITER
c
c     read in sizes
c
      PRINT *,'MAXX = (e.g. 64) '
      READ *,MAXX
      PRINT *,'MAXY = (e.g. 64) '
      READ *,MAXY
      ALLOCATE (F(MAXX,MAXY),DF(MAXX,MAXY))
C
      F = 2.
      F(:,MAXY) = 1.
      F(2:MAXX-1,2:MAXY-1) = 0.0
      ITER = 0
      FMAX = 1
      DO WHILE (FMAX .gt. 0.001)
         ITER = ITER + 1
         FORALL (J=2:MAXY-1, I=2:MAXX-1)
           DF(I,J) = (F(I,J+1) + F(I,J-1) +
     &                F(I-1,J) + F(I+1,J) ) * 0.25 - F(I,J)
         END FORALL
         FORALL (I=2:MAXX-1,J=2:MAXY-1)
            F(I,J) = F(I,J) + DF(I,J)
         END FORALL
         DF = ABS(DF)
         FMAX = MAXVAL (DF)
         PRINT *,'Iteration ',ITER,'  Max = ',FMAX
      END DO
      PRINT *, ITER, ' iterations needed'
      DEALLOCATE (DF, F)
      END
```

18.3 Using Overlapping

The following solution uses a help array with an overlap area. The great advantage is that the FORALL statement computing DF needs no temporaries and no further communication. Therefore this solution is much more faster than the previous ones.

```
      PROGRAM LAPLACE
      REAL F(:,:), DF(:,:), HF(:,:[1:1])
      INTEGER MAXX, MAXY
      REAL FMAX
      INTEGER ITER
c
c     read in sizes
c
      PRINT *,'MAXX = (e.g. 64) '
      READ *,MAXX
      PRINT *,'MAXY = (e.g. 64) '
      READ *,MAXY
      ALLOCATE (F(MAXX,MAXY),DF(MAXX,MAXY),HF(MAXX,MAXY))
C
      F = 2.
      F(:,MAXY) = 1.
      F(2:MAXX-1,2:MAXY-1) = 0.0
      ITER = 0
      FMAX = 1
      DO WHILE (FMAX .gt. 0.001)
         ITER = ITER + 1
         HF  = F             ! makes boundaries of F local
         FORALL (J=2:MAXY-1, I=2:MAXX-1)
```

```
      DF(I,J) = (HF(I,J+1) + HF(I,J-1) +
&                 HF(I-1,J) + HF(I+1,J) ) * 0.25 - HF(I,J)
    END FORALL
    FORALL (I=2:MAXX-1,J=2:MAXY-1)
       F(I,J) = F(I,J) + DF(I,J)
    END FORALL
    DF = ABS(DF)
    FMAX = MAXVAL (DF)
    PRINT *,'Iteration ',ITER,'  Max = ',FMAX
 END DO
 PRINT *, ITER, ' iterations needed'
 DEALLOCATE (HF, DF, F)
 END
```

References

[ALS91] E. Albert, J.D. Lukas, and G.L. Steele. Data Parallel Computers and the FORALL Statement. *Journal of Parallel and Distributed Computing*, 1(1):1–1, October 1991.

[Hig93] High Perforamnce Fortran Forum. High Performance Fortran Language Specification. Final Version 1.0, Department of Computer Science, Rice University, May 1993.

[Lov92] D. Loveman. Element Array Assignment - the FORALL Statement. In *Proceedings of Third Workshop on Compilers for Parallel Computers, Vienna Austria, July 6-9*, pages 109–120, 1992.

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

[WSG92] M.Y Wu, W. Shu, and Fox G.C. DO and FORALL: Temporal and Spatial Control Structures. In *Proceedings of Third Workshop on Compilers for Parallel Computers, Vienna, Austria, July 6-9*, pages 258–269, 1992.