ADAPTOR
Distributed Array Library (DALIB)
Version 4.0 (April 1996)

Thomas Brandes

Report Adaptor 4

March 27, 1996

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

ADAPTOR
Distributed Array Library (DALIB)
Version 4.0 (April 1996)

Thomas 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 developped 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.

The latest version incorporates the feedback of many users and contains already a lot of optimization techniques. Beside some restrictions, it supports full HPF including some features of the new proposed standard 2.0.

The HPF runtime system used for ADAPTOR is called DALIB (distributed array library). This library will be linked to the codes generated by the compiler. This document describes the design of this runtime system and the functions of the different modules.

1 Introduction

ADAPTOR (Automatic Data Parallelism Translator) is a system developed at the GMD for compiling data parallel High Performance Fortran [Hig93, KLS+94] (HPF) programs to equivalent Fortran 77 message passing programs. With the latest version, most features of HPF are supported [Bra96].

Beside the source-to-source translation (fadapt), the ADAPTOR system contains also a runtime system that will be linked with the generated programs (see Figure 1). The runtime system is a library called DALIB (distributed array library).

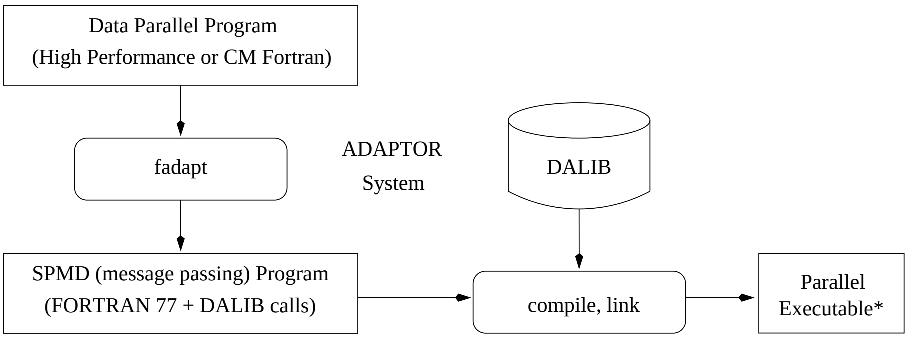

Figure 1: Overview of the ADAPTOR tool

1.1 Functionality of the DALIB

The DALIB provides the following functionality:

- handling of descriptors for processor arrays (topologies)
- handling of descriptors for arrays and sections of arrays
- handling of descriptors for the mapping of arrays
- computation of ownership and local sets
- computation of structured and unstructured communication schedules,
- low level communication (send, receive, wait, ...),
- high level communication (broadcast, reduction, barrier, ...),

- gathering and scattering of data
- indirect addressing of distributed arrays
- timing and tracing facilities,
- parallel random number generator,
- X-Windows interface.

The descriptors for arrays, sections and distributions have been introduced in version 3.0. This concept has been improved in such a way that the ADAPTOR tool can now deal with inherited mappings and dynamic redistributions.

1.2 Libraries

The translation of all modules results in one or two libraries that will be linked together with the generated programs of Adaptor.

- **dalib.a** is the library for the parallel node program.
- **unilib.a** is the library for the serial program.

1.2.1 Single Process Library

For many reasons it is useful to run HPF programs on single nodes (benchmarking, debugging). Therefore Adaptor cannot only generate parallel programs but also a serial version of the HPF program. The serial node program will be linked with a special library called *unilib*.

There is absolutely no communication for the uniprocessor version.

For timing there is a special module as the timing functions do not require processor synchronization.

A dummy module *never* became necessary, but functions of this module will never be called.

All modules will be put together to the library `unilib.a`.

1.2.2 Parallel Library

The parallel library contains also all modules from the serial library. Table 2 gives a short overview of all modules that are only part of the distributed array library.

1.3 Historical Information

Previous versions of ADAPTOR how also supported the host-node programming model for the generated message-passing programs. This is no longer the case. Only the SPMD model is available for the parallel execution.

It should be made sure that the source-to-source translation and the DALIB have the same version. Old versions of the runtime system will not work together with new versions of the source-to-source translation.

Module (filename)	short description	section
init1	initialization for single node process version	1.2.1
never	dummy routines for single node process version	1.2.1
timing1	timing functions, only single node	16
array	handling of descriptors for arrays	5
section	section descriptors	7
secarray	section/array descriptors	–
addressing	addressing of arrays/sections	6
memoryk	generic file for memory transfer functions	13.5
memgather	gathering of data	13.3
memscatter	scattering of data	13.3
memcopy	copying of data	13.2
memset	indirect access of data	13.4

Table 1: Modules of the UNILIB

2 Realization of the DALIB

2.1 FORTRAN Binding

The DALIB is written in C but will be linked to a Fortran program. Therefore some attention had to be paid to the language binding.

Unfortunately, this binding is machine-dependent. For a given subroutine call in Fortran,

```
INTEGER a
REAL b
....
CALL sub (a, b)
```

this subroutine must be defined in C in different ways:

```
void sub_ (int *a, float *b)    /* interface for SUN OS */
void sub  (int *a, float *b)    /* interface for IBM SP */
void SUB  (int *a, float *b)    /* interface for CRAY   */
```

Therefore the macro preprocessor `m4` is used to get machine-independent source files.

```
void FORTRAN(sub) (int a, float b)
```

The macro `FORTRAN` is defined for the different machines.

```
define(`FORTRAN',`$1')
define(`FORTRAN',`translit($1,abcdefghijklmnopqrstuvwxyz,
                              ABCDEFGHIJKLMNOPQRSTUVWXYZ)')
define(`FORTRAN',`$1_')
```

Module (filename)	short description	section
initall	initialization of parallel processes	3
init	machine dependent initialization of initall	3
arguments	getting the number of processes/processors	3
distribution	handling of descriptors for distributions	8
alignment	handling of descriptors for alignment	9
darray	handling of descriptors for distributed arrays	5
topology	definition of processor topologies	4
ddt	module for using derived data types	11.1
mailbox	primitive send and recv with mailboxes	11
pt2pt	machine dependent send and receive	11
groups	definition of processor groups	12.1
broadcast	broadcast from one node to all other nodes	12.2
reduction	reduction functions (sum, min, max, ...)	12.3
combiners	combiner functions for reductions	12.3
timing	timing functions	16
trace	trace utilities	16
assign	assignment of array sections	15.2
overlap	update of overlap areas	15.2
match	redistributions at subroutine boundaries	15.2
intersection	computing the intersection of regular sections	15.3
mapping	cyclic/block distribution of distributed dimensions	10.1
remote	remote addressing	6
schedule	managing structured schedules	15.5
structured	generating structured schedules for structured movements	15.4
pointers	handling of global addresses	19.2
unstructured	unstructured schedules	19.3
indirect0	indirect addressing of local arrays	18
indirect	indirect addressing of distributed arrays	19.4
random	random numbers	21.1
intrinsics	support of new intrinsic routines	
loccs	pipelined computation/communication	–
functions	new Fortran 90 intrinsices	21
hpf	some HPF intrinsics	21
grade	sorting for HPF libarary	21
scan	prefix/suffix operatons for HPF libarary	21
matmul	sorting for HPF libarary	21
shift	shifting of arrays using descriptors	17
shared	shared arrays	
shm	shared memory segments	

Table 2: Modules of the DALIB

2.2 Underlying Message Passing System

The DALIB is based on some simple routines for the point-to-point communication (see section 11). This communication can be realized very simply by using a any message passing system. The DALIB has been realized for different versions:

- One version is based on PVM. At the moment PVM version 3.1 and higher can be used. The DALIB and ADAPTOR can be used on all machines where PVM is running though this might not be the most efficient solution.
- One version is based on MPI. The DALIB for this new message passing interface will be optimized in the near future.
- One version is based on communication via a shared memory segment. This version can run on all systems that support these Unix System V features (e.g. SUN, KSR, Silicon Graphics).
- For the CM 5, Intel Paragon and IBM SP the library has been realized based upon the native communication routines (CMMD, NX/2 and POE).

In fact, exactly two modules are dependent on the message-passing system. These are the modules `init` for the initialization and `pt2pt` for the point-to-point communication.

If the used message passing system provides higher functionalities like MPI, some routines could be optimized. Good candidates are e.g. broadcasting and reductions on processor groups.

2.3 Shared Arrays

The DALIB supports shared or virtual shared arrays. The implementation uses System V shared memory segments.

All routines of the DALIB can be used in the same way for shared arrays (e.g. shifting, random numbers, structured communication).

2.4 One-sided Communication

For a given parallel loop all communication must be extracted before or after the parallel loop. Otherwise there is not the necessary synchronization to send explicitly required data.

There are some situations where the communication cannot be extracted. In this case there are two possibilities:

- the loop is executed serially, or
- the required communicaton is handled as one-sided communication.

The DALIB now supports one-sided communication based on remote procedure calls.

3 Parallel Processing Library and Initialization

3.1 Initialization

The module *init* contains subroutines for the initialization of parallel processing. This module is very machine-dependent.

```
void FORTRAN(dalib_init) ()      /* initialization for the SPMD model */
```

During the initialization the following steps are executed:

- the first node process creates other node processes (on some machines it is assumed that all processes are already loaded on the nodes),
- every process gets its own id and the number of existing processes (whoami),
- initialization of random area and mailboxes,
- definition of initial topologies (one-, two- and three-dimensional torus).

In the same way there exist some procedures for exiting of the parallel programs.

```
void FORTRAN(dalib_exit) ()
```

The by Adaptor generated programs will contain the corresponding calls for initialization and for finishing the parallel execution.

```
PROGRAM TEST                  PROGRAM TEST
! input program               ...
....                          CALL dalib_init ()
END                           ...
                              CALL dalib_exit ()
                              END
```

The SPMD parallel program contains the calls of `dalib_init` and `dalib_exit`.

Furthermore, there have been realized some functions to get internal information about processor ids and number of processes.

The first functions returns the processor id of the calling processor and the second one the total number of node processors that run the node program.

```
int FUNCTION(dalib_pid) ()
int FUNCTION(dalib_nproc) ()
```

The the node processes are numbered from 1 to p where p is the number of created node processes.

The following routine is responsibile for the machine-dependent initialization. Every process calls this routine at the beginning.

```
void dalib_machine_enroll (int *AllNP)

  - return in pcb.i the relative processor position
  - return in pcb.p the number of real existing processes
  - Process 0 or 1 returns in AllNP number of processes
    that will be used really (from environment, command line)
```

The following routine will be called by every process after the execution of the user program. In this routine it is possible to synchronisize and to collect runtime information or whatever else.

```
void dalib_machine_exit ()
```

3.2 Evaluation of Arguments

The module *arguments* provides a subroutine that determines the actual number of processors and the processor shaped.

```
void eval_arg (default_NP, max_NP, NP, trace_flag)
int default_NP, max_NP, *trace_flag;

/* default_NP : number of processors if no argument is specified
   max_NP     : maximal number of available processors
   trace_flag : returns 1 or 0 if trace should be done or not

   values of pcb (process control block) will contain processor shape
*/
```

4 Abstract Processor Arrays and Physical Processors

The physical processors are always considered as a linear set of processors, numbered from 1 to n.

A processor array (in the following also called topology) is the specification of a one-dimensional, two- or three-dimensional processor array. Such a toplogy is usually called a torus. A topology or torus cannot have more processors than are available. The topologies are used within HPF for multi-dimensional distributions of arrays.

Every abstract processor array is linearized and mapped to the physical processors, starting at a processor that can be specified explicitly.

The module *topology* realizes function for the management of default and new created abstract processor arrays.

4.1 Definition of Topologies

Every topology is identified by an internal integer value. If a topology is defined more than once, the same value is returned.

By default there are three internal topologies that will be defined during the initialization. The identifiers 1, 2 and 3 stand for these predefined internal topologies. These topologies are

NP	Topology 1	Topology 2	Topology 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 3: Default processor topologies in ADAPTOR

chosen in such a way that every processor is in every toplogy. This must not be the case with the other topologies.

```
void dalib_top_init ()
```

Table 3 shows some examples for initial topologies.

New topologies can be created with the following command:

```
void FORTRAN(dalib_top_create) (int *topid, int *first, int *rank,
                                int *n1, int *n2, int *n3)
```

- The output parameter `topid` is an integer value that is used for the identification of the topology.
- The input parameter `first` specifies which processor should be the first processor of the topology.
- The input parameters `n1`, `n2`, and `n3` specify the number of processors in the corresponding dimensions. If one value is less or equal 0, an appropriate choice for this values will be made automatically.

If a torus is created which needs more processors than are available, there will be a run-time error.

4.2 Queries for Processor Topologies

The following subroutines can be used to ask for processor ids in a given topology.

```
int dalib_top_pos (int topid, int pos[])
```

- The `topid` parameter specifies a defined topology
- The parameter array `p` specifies relative positions in the toplogy.
- The result is the wanted processor id.

The next subroutine gives processor identifiers of neighbored processors.

```
void dalib_top_neighbors (int *pleft, int *pright,
                          int topid, int dim     )
```

With the following subroutine one can ask for the extent of a dimension of a processor topology and for its own relative position in this topology. Every relative position is a value in the range from 1 to N.

```
void dalib_top_info (int topid, int dim, int *N, int *I)
```

The following functions are used to get information about the size and the ids of all processors in a toplogy.

- `dalib_top_size` returns the total number of processors in a topology

  ```
  int dalib_top_size (int topid)
  ```

- `dalib_top_elem` returns the id of the k-th processor in the topology `topid` ($0 \leq k < P$, P is the size of the topology).

  ```
  int dalib_top_elem (int topid, int k)
  ```

- `dalib_in_topology` returns 1 if the calling processor itself is in the topology, otherwise it returns 0.

  ```
  int dalib_in_topology (int topid)
  ```

- `dalib_top_dimpos` returns the processor id of the processor in the same topology as the calling processor, but at position `pos` for dimension `dim`.

  ```
  int dalib_top_dimpos (int topid, int dim, int pos)
  ```

- `dalib_top_relpos` returns the processor id of the processor in the same topology as the calling processor, but to its own position in dimension `dim` the value `pos` is added.

  ```
  int dalib_top_dimpos (int topid, int dim, int pos)
  ```

  The ids of the left and right neighbor in dimension `dim` are given by:

  ```
  p_left  = dalib_top_dimpos (topid, dim, -1)
  p_right = dalib_top_dimpos (topid, dim, 1)
  ```

- For every dimension of an abstract processor array, processors with the same position in the other dimensions are grouped together. The following function returns the group id to which the calling processor belongs (groups are described in section 12.1).

  ```
  int dalib_top_group (int topid, int dim)
  ```

4.3 Use of Processor Topologies

The default topologies are used for templates or arrays of the HPF program where no processor topology has been specified in the distribution directive. Therefore an array with three distributed dimensions is mapped to topology 3 by default.

New processor topologies will be created when using the processors directive of HPF.

```
!HPF$ PROCESSORS PA (2,2)
!HPF$ PROCESSORS PB (4)
!HPF$ PROCESSORS PC (3,0)
```

Processor array PA stands for a 2 x 2 torus, PB stands for a one-dimensional torus with four processors. If the parallel program is started with less than four processors, there will be a run-time error.

Processor array PC stands for a two-dimensional torus that has exactly 3 processors in the first dimension. The number of the processors in the second dimension depends on the number of processors that will be available at run-time. If 16 processors are available, PC stands for a 3 x 5 processor array.

Adaptor will generate the following calls to the DALIB:

```
      call dalib_top_create (PA_TOPID,1,2,2,2)
      call dalib_top_create (PB_TOPID,1,1,4)
      call dalib_top_create (PC_TOPID,1,2,3,0)
```

The most interesting feature of this module is that it allows to have different topologies that have no common processor. By this way, the DALIB can also be used for block-structured problems where arrays are mapped to different processors.

If there are 16 processors available, the following commands will define a one-dimensional torus with 4 processors, one with 8 processors and a two-dimensional torus with 4 processors. All processor arrays have no common processor.

```
!HPF$ PROCESSORS PA (1:4)
!HPF$ PROCESSORS PB (5:12)
!HPF$ PROCESSORS PC (13:14,2)

      call dalib_top_create (PA_TOPID,1,1,4)
      call dalib_top_create (PB_TOPID,5,1,8)
      call dalib_top_create (PC_TOPID,13,2,2,2)
```

```
         -------------------------
  PA     |  1  |  2  |  3  |  4  |
         -------------------------
         -------------------------------------------------
  PB     |  5  |  6  |  7  |  8  |  9  | 10  | 11  | 12  |
         -------------------------------------------------
         -------------
         | 13  | 15  |
  PC     -------------
         | 14  | 16  |
         -------------
```

5 Array Descriptors

5.1 Information of the Array Descriptor

An array decriptor is a pointer to a record that contains all relevant information regarding an array (rank, shape, distribution).

```
   typedef struct ArrayRecord *array_info;

struct ArrayRecord
 { int  ident;                       /* identification of this record   */
   int  size;                        /* size in bytes for one elem      */
   int  rank;                        /* rank of the array               */
   int  dynamic;                     /* DYNAMIC distribution/alignment  */
   char *data;                       /* pointer to the allocated data   */
   Distribution DistributeInfo;      /* describing the distribution     */
   Alignment AlignInfo;              /* describing the alignment        */
   Shared    SharedInfo;             /* describing how data is shared   */
   DimInfo   dimensions[MAX_DIMENSIONS]; /* info about the dims         */
 };

typedef struct
  { int global_size [2];
    int local_size  [3];
    int overlap [2];
  } DimInfo;
```

5.2 Descriptors for Sequential Arrays

The following routine defines a descriptor for an array. With the definition of a descriptor no memory for data is allocated.

```
   void FORTRAN(dalib_array_make_dsp)
          (array_info *array_id, int *rank, int *size)

   void FORTRAN(dalib_array_define)
          (array_info *array_id,
           int *lb1, int *ub1, ..., int *lb7, int *ub7)
```

As a descriptor needs internally some amount of memory, it should be freed after its use.

```
   void FORTRAN(dalib_array_free) (array_info *array_id)
```

Every internal descriptor must also contain information about overlap areas. Therefore the size of this area must also be defined for the descriptor. This routine must not be called if there is no overlap area at all.

```
   void FORTRAN(dalib_array_overlap)
       (array_info *array_id,
        iint *lb1, int *ub1, ..., int *lb7, int *ub7)
```

```
void dalib_array_create_copy (new_array_id, old_array_id)
```

If a descriptor has been defined, this descriptor can be used to allocate memory for the array.

```
void dalib_array_allocate_ (array_id, a_ptr, a_size,
                            a_zero, a_dim1, a_dim2, a_dim3,
                            a_dim4, a_dim5, a_dim6, a_dim7)
```

```
   real A (:,:,:)
   allocate (A(N1,2:N2,0:N3))
   A = 0.0
   deallocate (A)

   call dalib_array_create (A_DSP,3,1,N1,1,49,1,11)
   call dalib_array_allocate (A_DSP,A,4,A_ZERO,A_DIM1,A_DIM2,A_DIM3)
   DO I3=1,11
      DO I2=1,49
         DO I1=1,N1
            A(A_ZERO+I3*A_DIM2+I2*A_DIM1+I1) = 0.0
         END DO
      END DO
   END DO
   call dalib_array_free (A_DSP)
```

```
void dalib_array_access (array_id, a_ptr, a_size,
                          a_zero, a_dim1, a_dim2, a_dim3,
                          a_dim4, a_dim5, a_dim6, a_dim7)
```

6 Addressing of Arrays and Sections

The module *addressing* provides the necessary routines to compute positions of elements in arrays and sections.

The main routine `dalib_array_addressing` returns for a given array (specified by its descriptor) and a given processor id the following values:

- a pointer `data` to the first allocated element on the processor,
- an offset `first` for the fictive zero element (this means data - first * size is the pointer to `A(0,...,0)`
- an array `total` of values for every dimension where `total[i]` specifies the difference between two succesive elements in dimension i.

```
dalib_array_addressing (array_info array_id, int pid,
                        char **data, int *first, int total[])
```

The address to a certain array element

```
A(I1, I2, I3, ..., Ik)
```

can be computed in the following way:

```
int first, total[k];
char *data;
...
dalib_array_addresing (array_id, my_id, &data, &first, total);
offset = -first;
for (i=0; i<k; i++)
    offset += total[i] * I(i+1);

/*  data + offset * array_id->size is the pointer to A (I1, ..., Ik) */
```

Internally, the following two routines are provided:

```
dalib_array_local_addressing (array_info *array_id,
                              int *first, int total[])

dalib_array_global_addressing (array_info *array_id,
                               int *first, int total[])
```

For distributed arrays, the local addressing is used. Only for shared or virtual shared arrays the global addressing can be used.

Usually, not only single elements are addressed. The following routines return derived data types that stand for the local part of an array or of a section for the called processor (the derived data types can be used for a lot of other operations, see section 11.1).

```
void dalib_make_array_ddt (dd_type *new_ddt, array_info array_id)

void dalib_make_perm_section_ddt (dd_type *new_ddt,
                               section_info section_id, int perm[])
```

The routine for sections has an additional parameter that specifies a permutation of the dimensions.

```
A(1:3,4:6)
```

Instead of the sequence:

```
A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2), A(1,3), A(2,3), A(3,3)
```

we have

```
A(1,1), A(1,2), A(1,3), A(2,1), A(2,2), A(2,3), A(3,1), A(3,2), A(3,3)
```

7 Section Descriptors

7.1 Information of the Section Descriptor

Array operations cannot only be called with full arrays but also with sections of an array. For this purpose, the runtime system provides descriptors for sections.

```
typedef struct SectionRecord *section_info;
#define NO_SECTION ((section_info) 0)
```

A section is characterized by its global range for every dimension. If the dimension is distributed, the descriptor contains also the local range for this dimension.

```
typedef struct
  { int is_range;           /* 0 stands for single element   */
    int global_range [3];
    int local_range  [3];
  } SecDimInfo;

struct SectionRecord

 { int ident;              /* identification of this record */
   array_info array_id;    /* pointer to the full array descriptor */
   SecDimInfo dimensions[MAX_DIMENSIONS]; /* info about the section dims   */
 };
```

The following predicate returns true if the pointer stands really for a section descriptor. For this predicate the component `ident` of the `SectionRecord` is used that must have a very special value.

```
int dalib_is_section_info (section_info section_id)
```

For debugging and error messages section descriptors can be printed.

```
void dalib_print_section_info (section_info section_id)
```

7.2 Operations for Section Descriptors

```
void FORTRAN(dalib_section_create) (section_id, array_id,
                            kind1, lb1, ub1, inc1,
                            ...
                            kind7, lb7, ub7, inc7)

void FORTRAN(dalib_section_free) (section_info *section_id)

 call dalib_section_create (ISEC1_DSP, A_DSP, 0, 1, 1, 1, 1, 2, 5, 1)
 call dalib_section_free (A_DSP)

void FORTRAN(dalib_section_point) (section_id, array_id,
                        ind1, ind2, ind3, ind4, ind5, ind6, ind7)
```

The following operation creates of an array descriptor a corresponding section descriptor that stands for the full array. This operation is required if routines are called that can only deal with section descriptors.

```
void dalib_section_full (section_info *section_id,
                         array_info   array_id)
```

The following routine recomputes the local size of a section by the global size and the given distribution.

```
void dalib_section_reset (section_info section_id)
```

The following routines give information about the section.

```
int dalib_section_rank (section_info section_id)

void dalib_section_global_shape (section_info section_id, int *rank,
                                  int lb[], int ub[], int stride[] )

void dalib_section_local_bounds (section_info section_dsp, int dim,
                                  int *lb, int *ub, int *inc)

void dalib_section_local_shape (section_info section_id,
                                 int *rank, int lb[], int ub[])
```

8 Distribution Descriptors

8.1 Structure of the Descriptors

For the distribution of a dimension there exist four possibilities:

- the dimension is a serial dimension

```
typedef struct { short axis_type;
               } ySERIAL_DIM;
```

- the dimension is block distributed (BLOCK(n))

```
typedef struct { short axis_type;
                 int   axis_info;      /* BLOCK (axis_info)           */
                 int   axis_map;       /* dimension of processor array */
               } yBLOCK_DIM;
```

- the dimension has different block sizes (BLOCK(array)) where the descriptor will contains an array for the offsets

```
typedef struct { short axis_type;
                 int  *axis_offsets;   /* BLOCK (axis_info_array)     */
                 int   axis_map;       /* dimension of processor array */
               } yVBLOCK_DIM;
```

- the dimension is cyclic distributed (CYCLIC(n))

```
typedef struct { short axis_type;
                 int   axis_info;       /* CYCLIC (axis_info)           */
                 int   axis_map;        /* dimension of processor array */
               } yCYCLIC_DIM;
```

The descriptor for the distribution of one dimension is the union of the different structures.

```
typedef union {
 short axis_type;
 ySERIAL_DIM     SERIAL_DIM;
 yBLOCK_DIM      BLOCK_DIM;
 yVBLOCK_DIM     VBLOCK_DIM;
 yCYCLIC_DIM     CYCLIC_DIM;
 } DistDimDescriptor ;       /* info about the distribution of one dim */
```

The distribution of an array is then described by the identifier of the topology and by a descriptor for every dimension.

```
struct DistributionRecord {
 int topology;
 int no_aligned;
 int max_aligned;
 array_info *aligned_arrays;   /* aligned_arrays [max_aligned] */
 DistDimDescriptor dims[MAX_DIMENSIONS];
 };
```

8.2 Creation of Distributions

```
void FORTRAN(dalib_distribute) (array_id, topology, type1, info1,
                        type2, info2, type3, info3, type4, info4,
                        type5, info5, type6, info6, type7, info7)
array_info *array_id;
int *toplogy;
int *type1, ..., *type7;
int *info1, ..., *info7;

void dalib_dist_free (Distribution dist)
```

8.3 Informations about the Distribution

```
void dalib_distribution_info (array_info array_id,
                              array_info *template_id, int *top_id)

void dalib_distribution_dim_info
           (array_info array_id, int dim,
            int *topology_id, int *topology_dim,
            int *base, int *stride, int *lb, int *ub,
            int *kind, int **offsets)

int dalib_distribution_topology (array_info array_id)
```

```
void dalib_distribution_top_query
            (array_info array_id, int top_dim,
             int *kind, int *index_dim, int *top_pos)

void dalib_distribution_data (array_info array_id,
                              int type[], int info[], int map[])
```

8.4 Computing Offsets for Irregular Block Sizes

The following table demonstrates the distribution of a dimension of the size `1:50`.

	0	1	2	3	4	5	NP-1	NP
sizes	5	6	7	8	9	10	11	
axis offsets	0	5	11	18	26	35	45	56
axis offsets "	1	6	12	19	27	36	46	51
ranges	1:5	6:11	12:18	19:26	27:35	36:45	46:50	

8.5 Miscallaneous Routines

The following routine returns a serious error if array_id has not a distribution.

```
void dalib_distribution_check (array_info array_id)

void dalib_distribution_print (Distribution dist, int rank)
```

8.6 Aligned Arrays

For a template or array that is distributed we have to know the arrays that are aligned to it.

```
void dalib_set_aligned_to (array_info alignee, array_info template)
void dalib_set_dealigned  (array_info alignee, array_info template)

void dalib_get_aligned_arrays (array_info template_id,
                               int *n, array_info **alignees)
```

9 Alignment Descriptors

9.1 Structure of the Descriptors

A dimension of the alignee (source array) can be aligned in the following ways:

- it can be collapsed and is a serial dimension

  ```
  typedef struct { short axis_type;
                 } ySERIAL_DIM;
  ```

- it can be aligned with a linear function

```
typedef struct { short axis_type;
                 short axis_map;        /* dimension of template          */
                 int   base, stride;    /* align I with base + I * stride */
               } yALIGNED_DIM;
```

- it can be mapped

```
typedef struct { short axis_type;
                 short axis_map;        /* dimension of template     */
               } yMAPPED_DIM;           /* align I with I, or (:)    */
```

The descriptor for a source dimension is the union of the three structures.
typedef union

```
 short axis_type;
 ySERIAL_DIM     SERIAL_DIM;     /* align (...,*,...)  with ...              */
 yALIGNED_DIM    ALIGNED_DIM;    /* align (I)  with T (..,base+stride*I,..) */
 yMAPPED_DIM     MAPPED_DIM;     /* align (...,:,...)  with T (...,:,...)   */

 } AlignSourceDescriptor ;
```

A dimension of the align-target can be:

- a dimension of the alignee is mapped to it

```
typedef struct { short axis_type;
                 short axis_map;        /* dimension of source array */
               } ySOURCE_DIM;
```

- an embedded dimension

```
typedef struct { short axis_type;
                 int   base;
               } yEMBEDDED_DIM;         /* align ... T(..,2,...)     */
```

- a replicated dimension

```
typedef struct { short axis_type;
               } yREPLICATED_DIM;
```

A descriptor is the union of the three different kinds:

```
typedef union {

 short axis_type;

 ySOURCE_DIM     SOURCE_DIM;     /* align (....,I,...) with T (...,f(I),...) */
 yREPLICATED_DIM REPLICATED_DIM; /* align  ...         with T (...,*,...)    */
 yEMBEDDED_DIM   EMBEDDED_DIM;   /* align  ...         with T (...,val,...)  */

 } AlignTargetDescriptor ;
```

A descriptor for an alignment describes how the dimensions of the alignee are mapped to it.

```
struct AlignmentRecord

 { array_info            template;
   AlignTargetDescriptor target[MAX_DIMENSIONS];  /* rank of template   */
   AlignSourceDescriptor source[MAX_DIMENSIONS];  /* rank of array      */
 } ;
```

9.2 Definition of Alignments

The following test can be used to check whether an array is aligned or not.

```
  array_id->AlignInfo == NO_ALIGNMENT

  void dalib_align_source
      (array_info *array_id, array_info *temp_id,
       type1, dim1, base1, stride1,
       ...
       type7, dim7, base7, stride7)

  int *type1, ..., *type7;
  int *dim1, ..., *dim7;
  int *base1, ..., *base7;
  int *stride1, ..., *stride7;

  void dalib_align_target
      (array_info *array_id, array_info *temp_id,
       type1, info1, ..., type7, info7)

  int *type1, ..., *type7;
  int *info1, ..., *info7;
```

9.3 Informations about the Alignment

The routines for asking about alignments are very similar to the routines for distributions (see section 8.3).

```
   void dalib_alignment_info (array_info array_id,
                              array_info *template_id, int *top_id)

   void dalib_alignment_dim_info
             (array_info array_id, int dim,
              int *topology_id, int *topology_dim,
              int *base, int *stride, int *lb, int *ub,
              int *kind, int **offsets)
```

Due to the alignment we can have replication of arrays. The following routines are used to get information about it.

```
int dalib_alignment_is_replicated (array_id)

void dalib_alignment_rep_dims (array_info array_id,
    int *top_id, int *number, int rep_dims[])

 int dalib_distribution_topology (array_info array_id)

 void dalib_distribution_top_query
             (array_info array_id, int top_dim,
              int *kind, int *index_dim, int *top_pos)
```

The routine might return:

- if kind is `kSOURCE_DIM`, index_dim is mapped to this topology dimension,
- if kind is `kREPLICATED_DIM` we have a replicated dimension
- if kind is `kEMBEDDED_DIM` top_pos is the owner of this embedded dimension

The following routine gives information about the target of the alignment (needed for HPF library routine `HPF_TEMPLATE`).

```
void dalib_align_target_data (array_info array_id,
                              int type[], int info[])
```

The next routine gives information how the alignee is aligned to its target (needed for HPF library routine `HPF_ALIGNMENT`).

```
void dalib_align_source_data (array_info array_id,
                              int map[], int base[], int stride[])
```

9.4 Miscallaneous Routines

```
void dalib_alignment_check (array_info array_id)
```

10 Computing Local Sizes of Distributed Arrays

This section describes the modules of the DALIB that are used for the definition of the mapping of arrays to processor topologies.

Let A be an array variable of the program and I_A the index set of possible subscripts, P the set of node processes. Then the distribution can formally be described by:

$$d_A : I_A \rightarrow P$$

$d_A^{-1}(p)$ is the data domain of variable A for process p.

Automatic partitioning tools like Adaptor have to take care about such mappings. While the user describes what kind of mapping is chosen, the tool has to know how to distribute the data and which data resides on which processors.

- The module *mapping* computes local ranges of distributions and offers functions to compute the owners of single elements.
- The module *manager* has similar functions, but here are the functions are based on a dimension of a processor topology (see section 4).

10.1 Distribution of a Dimension

Let A be a one-dimensional array of N elements that is mapped to M processes. Then the distribution of A is as follows (block mapping to rectangular segments):

$$I_A = \{1, ..., n\} \qquad P = \{1, ..., m\}$$

$$d_A \quad : \quad \text{i} \mapsto \text{(i*m-1) div n + 1}$$

$$d_A^{-1}(p) = \{ \text{ (p-1)*n div m + 1, ..., p*n div m } \}$$

If n is a multiple of m (n = x * m) the formulas become simpler:

$$d_A \quad : \quad \text{i} \mapsto \text{(i-1) div x + 1}$$

$$d_A^{-1}(p) = \{ \text{ (p-1)*x + 1, ..., p*x } \}$$

```
n = 17  m = 5   A(1:3)  A(4:6)  A(7:10)  A(11:13)  A(14:17)
                 P(1)    P(2)    P(3)      P(4)     P(5)

n = 16  m = 4   A(1:4)  A(5:8)  A(9:11)  A(12:16)
                 P(1)    P(2)    P(3)     P(4)
```

The module *mapping* offers routines for such distributed dimensions. The subroutines of this module are only used internally within the DALIB for determing the local parts of distributed dimensions.

```
void dalib_block_size (int P, int I, int lb, int ub,
                       int *low, int *high, int *size)

void dalib_cyclic_size (int P, int I, int lb, int ub
                        int *low, int *high, int *size)
```

- The input parameter `P` stands for the number of processors available for the distribution of the dimension.
- The input parameter `I` gives the position of a processor ($1 <= I <= P$).
- The input parameters `lb` and `ub` specify the range of the distributed dimension.
- The output parameter `low`, `high` give the first and the last index that is located on processor `I`. For block distributions that is exactly the local range, for cyclic distributions it has to be observed that only every P th element of this range is local on processor `I`.
- The output parameter `size` specifies additionally the number of local elements on the processor.

- The subroutines allow processors not only to ask for their local range but also for local ranges of other processors.

In case of an irregular block distributed dimension, the `offsets` array can directly be used to determine the local size (see section 8.4).

```
low  = offsets[I-1];
high = offsets[I]-1;
size = high - low + 1;
```

The following functions are used to find the owner of an index `k` for a dimension with N elements that is distributed on `P` processors.

```
int dalib_block_owner (int P, int lb, int ub, int k)

int dalib_cyclic_owner (int P, int N, int k)
```

For irregular block sizes the owner is determined via the array `offsets` that has $P+1$ elements. The element `offset[I]` indicates the first index on processor `I+1`.

```
int dalib_vblock_owner (int offsets[], int P, int k)
```

For many situations it is important to find out the local part of a global section. If there are no block-cyclic distributions, the local part is also a section again (intersection of sections is a section again, see also section 15.3)

```
 low:high:stride   -> loc_low:loc_high:loc_stride

void dalib_block_range (int P, int I, int lb, int ub,
                        int low, int high, int stride,
                        int *loc_low, int *loc_high, int *loc_stride)

void dalib_vblock_range (int offsets[], int I,
                        int low, int high, int stride,
                        int *loc_low, int *loc_high, int *loc_stride)

void dalib_cyclic_range (int P, int I, int lb, int ub,
                         int low, int high, int stride,
                         int *loc_low, int *loc_high, int *loc_stride)
```

For efficiency reasons, similar routines are provided for the special case where the stride is 1.

```
void dalib_block_slice (int P, int I, int lb, int ub,
                        int low, int high,
                        int *loc_low, int *loc_high)

void dalib_cyclic_slice (int P, int I, int lb, int ub,
                         int low, int high,
                         int *loc_low, int *loc_high)
```

10.2 Managing Array Distributions

The module *darray* (distributed arrays) exports subroutines that can be called from FORTRAN programs.

Testing for ownership provided the following function.

```
int dalib_is_local (array_info array_id, int *dim, int *index_val)
```

The following routines are used for getting the local part of a global range.

```
 global range :  global_lb : global_ub : global_stride
 local  range :  local_lb  : local_ub  : local_stride

void dalib_array_lrange (array_info *array_id, int *dim,
                         global_lb, global_ub, global_stride,
                         local_lb, local_ub, local_stride)

void dalib_array_lslice (array_info *array_id, int *dim,
                         global_lb, global_ub,
                         local_lb, local_ub)

/* mrange : stride * I + base  -> [glb:gub:gstride] */

void dalib_array_mrange (array_info *array_id, int *dim,
                         int *base, int *stride,
                         global_lb, global_ub, global_stride,
                         local_lb, local_ub, local_stride)

void dalib_array_info (array_info array_id,
                       array_info *template_id, int *top_id)

void dalib_array_dim_info
           (array_info array_id, int dim,
            int *topology_id, int *topology_dim,
            int *base, int *stride, int *lb, int *ub,
            int *kind, int **offsets)
```

The following routine asks only for the topology dimension to which the array dimension is mapped to.

```
void dalib_array_map_query (array_info array_id, int array_dim,
                            int *top_id, int *top_dim)
```

The next routine is similar but asks also for the special position of one array element.

```
void dalib_array_topmap (array_info array_id, int array_dim, int val,
                         int *top_id, int *top_dim, int *top_pos)

int dalib_distribution_topology (array_info array_id)
```

```
void dalib_array_top_query
            (array_info array_id, int top_dim,
             int *kind, int *index_dim, int *top_pos)
```

The following predicated returns true if the array is replicated.

```
int dalib_is_replicated (array_info/section_inf
```

The following routine provides consistency between all incarnations of a replicated array.

```
void dalib_replicate (section_info section_id)
```

If a scalar value of a distributed array is needed, the owner of this element must be fixed and afterwards the scalar value can be broadcast to all other processors.

```
void dalib_find_owner (array_info *array_id, int *nr
          int *dim1, int *index1,
          int *dim2, int *index2,
          int *dim3, int *index3)
```

- fix the owner that owns certain elements of the array array_id
- nr specifies how many arguments have to be used
- dim1, ..., dim3 specify distributed dimensions of the array
- index1, .., index3 are corresponding index values
- Result: owner fixes one processor that owns the searched element

If the owner has been fixed, the following routine can be used to broadcast data (which must be contiguous) by the owner:

```
void dalib_node_get (char *rep_data, int *size)
```

```
int dalib_is_aligned (section_info section1, section_info section2)
```

- A dimension with `N` elements will be distributed on dimension `dim` of the processor array specified by `topid`.
- The range `low:high:stride` is the local range on the calling processor for this dimension. The number of local elements is given by `size`.

11 Point-to-Point Communication

The module *mailbox* realizes the point-to-point communication at the machine-independent level where the module *pt2pt* realizes the machine-dependent primitives for blocking and non-blocking send and receive.

11.1 Derived Data Types

Very similar to the concept of derived data types in the MPI standard [Mes94], the DALIB uses internally data structures that describe a sequence of data in the users memory area. These data structures and operations on them are defined in the module *ddt*.

The importance of derived data types is given by the fact that they can be used for sending and receiving. Therefore the packing and unpacking of data is left to the communication system itself. Derived data types are very helpful in the context of non-blocking communication as they specify what has to be done with the received data.

Within a runtime system for an HPF compiler these derived data types describe subsections of arrays and indirect addressed arrays.

A derived data type is a pointer to an internally defined structure. The null pointer is used for specifying an empty derived data type.

```
typedef union ddt_info *dd_type;

#define NO_DDT ((dd_type) 0)
```

A derived data type can be defined as a contiguous memory section, as a section, as an indirect addressed memory area and as a union of them.

```
void dalib_ddt_def_simple (dd_type *new_ddt, char *memory, int size)

  - contiguous section in memory of rank 0

void dalib_ddt_def_section (dd_type *new_ddt, char *memory, int size,
                            int rank, int inc1, n1, ..., inc7, n7)

  - section memory [k1*inc1+...+k7*inc7], 0 <= ki < ni
    (one element has size bytes)

void dalib_ddt_def_indexed (dd_type *new_ddt, int no_elems,
                            int *indexes                  )

  - dates are  memory [indexes[i]], 0 <= i < no_elems
    (memory and size will be set later)

void dalib_ddt_set_data (dd_type ddt, char *data, int size)

void dalib_ddt_def_combine (dd_type *ddt, dd_type ddt1, dd_type ddt2)

  - build the union of two derived data types
```

A derived data type is an internal data structure that needs some memory space. Therefore it must be freed explicitly after its use. Usually this is done after the sending and receiving of a message. Within the communication system it might be necessary to copy the ddt, especially for non-blocking communication. Instead of copying, an internal reference counter is used.

```
void dalib_ddt_setref (dd_type ddt)
```

```
    - set an additional reference to the ddt structure

  void dalib_ddt_free (dd_type ddt)

    - free memory used for ddt if no other references
```

The following routines can be used to ask for some properties of a derived data type.

```
  void dalib_ddt_get_size (dd_type ddt, int *size)

    - returns size in bytes of the memory ared defined by ddt

  void dalib_ddt_is_contiguous (dd_type ddt, int *is, char **address)

    - returns true for is if ddt specifies contiguous memory section
      (address is pointer to begining of this section)
```

The following routines are used to pack or unpack or copy data specified by derived data types. This might be necessary for sending and receiving.

```
  void dalib_ddt_pack (char *buffer, dd_type ddt)

    - contiguous buffer is filled with data from memory def. by ddt

  void dalib_ddt_unpack (dd_type ddt, char *buffer, int op)

    - memory (def by ddt) is filled by contigous buffer
      (op specifies the operation applied with old value)

  void dalib_ddt_move (dd_type ddt_target, dd_type ddt_source)

    - copies data from one memory area to another memory area
```

11.2 Sending and Receiving

One central point of this library is that every process should know from which process it receives a message. Furthermore, there are no tags to distinguish messages.

```
void dalib_send           (int to,   char *message, int length)
void dalib_receive        (int from, char *message, int length)
void dalib_unpack_receive (int from, char *message, int length)
```

11.3 Blocking Communication

The machine-dependent realization of the blocking communication must have the following properties:

- after sending the message the sender can rewrite the memory area of the message

- if one process sends more messages to one other process, these messages arrive in the same order as they have been sent.
- if the message cannot be delivered directly the message is buffered

```
void mpi_send (int to, char *msg, int len, int kind)
void mpi_receive (int from, char *msg, int len)
void mpi_sendreceive (int to, char *out_msg, int out_len,
                      int from, char *in_msg, int in_len)
```

Usually these function can be realized directly upon the communication functions of the underlying hardware architecture, e.g. on the iPSC/860 with csend and crecv. The tag field should be used to distinguish the messages from different processors.

The realization for the Alliant FX/2800 based on a shared memory region has the following properties:

- every process has a mailbox that is a private memory area
- if a process sends a message this message is put in the mailbox of the receiver
- sending is not blocking as long as the receiving process has not a pending message from the same process
- no process can put two messages in the same mailbox (it will wait until the first message has been read)
- in the mailbox is always a copy of the message, so the sending process can use the memory of the sent data
- if the mailbox is full, automatic garbage collection will be initiated (system stop if no more place will be available)

11.4 Non-Blocking Communication

In an advanced modus ADAPTOR can take advantage of overlapping the communication with computations. This modus requires the realization of non-blocking communication.

```
int mpi_isend (int to, char *msg, int len)
int mpi_ireceive (int from, char *msg, int len)
int mpi_done (int msg_id)
void mpi_wait (int msg_id)
```

12 Collective Communication

The DALIB supports collective communication like broadcast and reductions for processor groups. A processor group is a selected set of processors, usually all node processors or all processors that belong to a specific dimension of a processor topology.

12.1 Processor Groups

The following function creates a processor group that contains the processes with the ids $first$, $first + step$, ..., $first + (size - 1) * step$. The function returns an integer value that identifies exactly this group.

```
int dalib_group_create (int size, int first, int step)
```

The initialization routine of the module creates two default groups, one group that contains the NP node processes and one group that cotains all node processes and the host process.

```
int dalib_groups_init (int NP)
```

The following two function returns the integer identification of these two groups. Please note that for the *ONLY-NODE* programming model these two groups are identical.

```
int dalib_group_all ()
int dalib_group_nodes ()
```

After the creation or the definition of groups these groups can be used for collective communication.

```
void dalib_group_bcast (char *data, int dlen, int source,
                        int group_id                    )

void dalib_group_reduce (char *data, int dlen, func, int group_id)
```

It should be mentioned that groups are only used internally. For reductions and broadcasting at the FORTRAN level topologies and dimensions are used instead of groups.

12.2 Broadcasting

With broadcast values of nodes and host can be exchanged. The broadcast subroutines are part of the module *broadcast*.

```
void dalib_broadcast_ (char *data, int *size, int *sendpid)
```

- The parameter `data` is the pointer to the data that will be sent or received and the parameter `size` is the size of the data in bytes.
- The parameter `sendpid` specifies the processor that has the legal value of the data. The value of its data is broadcast.

There is also a function for broadcasting data along one dimension of a processor topology. For this routine it is assumed that the first processor of the specified dimension in the topology has the actual value.

```
void dalib_top_broadcast (int topid, int dim, char *data, int size)
```

This routine could be used for a full broadcast within a topology in the following way:

```
void dalib_fulltop_broadcast (int topid, char *data, int size)

{ int i, rank;
  if (!dalib_in_topology (topid)) return;
  rank = dalib_top_rank (topid);
  for (i=1; i<= rank; i++)
     dalib_top_broadcast (topid, i, data, size);
} /* dalib_fulltop_broadcast */
```

This solution might be inefficient as also some illegal values are broadcast.

12.3 Reductions

The module *reduction* realizes different reduction operations for all node processors. The module *combiners* contains local reduction operations for two single elements. The local reduction operations will also be used for global send operations (see section 19.4).

Every node has exactly one value that is usually the result of the local reduction on that processor. The local results can be combined to a global result with the following reduction subroutine:

```
void dalib_reduction_ (char *dat, int *op)
```

The second parameter of the subroutine specifies the kind of reduction that is executed on the data.

```
 1 : min_ints          /* MINVAL,  integer*4  */
 2 : min_reals         /* MINVAL,  real*4     */
 3 : min_doubles       /* MINVAL,  real*8     */
 4 : max_ints          /* MAXVAL,  integer*4  */
 5 : max_reals         /* MAXVAL,  real*4     */
 6 : max_doubles       /* MAXVAL,  real*8     */
 7 : add_ints          /* SUM,     integer*4  */
 8 : add_reals         /* SUM,     real*4     */
 9 : add_doubles       /* SUM,     real*8     */
10 : mult_ints         /* PRODUCT, integer*4  */
11 : mult_reals        /* PRODUCT, real*4     */
12 : mult_doubles      /* PRODUCT, real*8     */
13 : and_ints          /* AND,     integer*4  */
14 : or_ints           /* OR,      integer*4  */
15 : eor_ints          /* EOR,     integer*4  */
16 : and_bools         /* ALL,     logical*4  */
17 : or_bools          /* ANY,     logical*4  */
18 : neq_bools         /* PARITY,  logical*4  */
19 : add_complexes     /* SUM,     complex*8  */
```

After the execution of the procedure every node process has the global result in the corresponding variable dat. This is also true for the host process whose old value is not used for the global reduction operation.

The following subroutine is used when it is necessary to remember the processor id where the minimum or maximum value appears.

```
void dalib_pos_reduction_ (char *dat, int *op)

  1 : min_ints           /* MINVAL,  integer*4  */
  2 : min_reals          /* MINVAL,  real*4     */
  3 : min_doubles        /* MINVAL,  real*8     */
  4 : max_ints           /* MAXVAL,  integer*4  */
  5 : max_reals          /* MAXVAL,  real*4     */
  6 : max_doubles        /* MAXVAL,  real*8     */
```

When one of these functions has been used, the following procedure can be used to get values from the processor that has had the minimum or maximum value.

```
void dalib_loc_exchange_ (char *data, int *size)
```

These functions are used to realize global operations on arrays. The latter ones are used for minloc and maxloc operations.

```
real A(:), S
integer P
...
S = sum (A)
P = minloc (A)
```

A reduction is split up in a local reduction and a global reduction that needs communication between the different processes.

```
 ! host program                    ! node program

 real S, MIN_A                     real S, A(:), MIN_A
 integer P                         integer P
 ...                               ...
 call dalib_reduction (S,8)        S = sum (A)      ! local sum
                                   call dalib_reduction (S,8)
 call dalib_pos_reduction          MIN_A = minval (A)
$     (MIN_A, 4)                   call dalib_pos_reduction (MIN_A, 4)
                                   P = minloc (A)
 call dalib_loc_exchange (P,4)     call dalib_loc_exchange (P,4)
```

13 Subroutines for Memory Transfer

Array sections are a central data structure within data parallel programs. Many array operations are based on sections (copy of sections, transpose, sending and receiving), especially in High Performance Fortran programs.

The DALIB offers many functions for operations on such sections. This section describes the necessary subroutines within the local memory.

- The module *memcopy* has functions for copying a section.
- The module *memgather* gathers data of a section to a contiguous memory section.
- The module *memscatter* scatters data of a contiguous memory section into an array section.
- The module *memset* offers operations for indirect addressing of local data.

13.1 Description of Sections

Usually an array section is described by the number of bytes for one element and by the ranges in every dimension.

```
real A (N1, N2, ..., Nk)
...
A(lb1:ub1:str1,lb2:ub2:str2,...,lb_k:ub_k:str_k)

n1, str1, n2, str2 * N1, n3, str3 * N1 * N2
```

Such a section will be described by

- A(lb1,lb2,...,lbk) is the address of the first element
- n1, ..., nk are the number of elements in the sections
- stride1, ..., stridek are the number of bytes between two elements of the section with respect to dimension k

$$mem_{A,n1,str1,...,nk,strk} = \{A + i_1 * str_1 + ... + i_k * str_k, 0 <= i_j < n_j\}$$

Furthermore, this representation has the following advantages:

- dimensions with only a single element have not to be considered,
- transposed array sections can also be represented (by switching the corresponding parameters).

A typical loop that operates on such a section has the following shape:

```
for (i<k> = n<k>; i<k>--; source += stride<k> - n<k-1>*stride<k-1>)
   ...
      for (i3 = n3; i3--; source += stride3 - n2 * stride2)
        for (i2 = n2; i2--; source += stride2 - n1 * stride1)
          for (i1 = n1; i1--; source += stride1)
              /* operate on source */
```

13.2 Copying and Moving of Data

The following subroutines of the module *memcopy* are used for copying array sections.

```
void dalib_memcopy (target, source, size)
void dalib_move1d (target, source, size, stride1, n1)
void dalib_move2d (target, source, size, stride1, n1, stride2, n2)
...
void dalib_move7d (target, source, size, stride1, n1, ..., stride7, n7)

unsigned char *target, *source;
int size, stride1, n1, ..., stride7, n7;
```

13.3 Gathering and Scattering of Data

The following subroutines of the modules *memgather* and *memscatter* are used to compress the data of an array section to a contiguous memory section and vice versa.

```
void dalib_gather1d (target, source, size, stride1, n1)
void dalib_gather2d (target, source, size, stride1, n1, stride2, n2)
...
void dalib_gather7d (target, source, size, stride1, n1, ..., stride7, n7)

void dalib_scatter1d (target, source, size, stride1, n1)
void dalib_scatter2d (target, source, size, stride1, n1, stride2, n2)
...
void dalib_scatter7d (target, source, size, stride1, n1, ..., stride7, n7)

unsigned char *target, *source;
int size, stride1, n1, ...., stridek, nk;
```

13.4 Indirect Addressing

The subroutines of the module `memset` are used for indirect addressing of memory.

```
void dalib_memget  (target, source, index, n, size)
void dalib_memgetp (target, source, index, n, size)
void dalib_memgetm (target, source, index, mask, n, size)

void dalib_memset  (target, index, source, n, size)
void dalib_memsetp (target, index, source, n, size)
void dalib_memsetm (target, index, source, mask, n, size)

unsigned char *target, *source;
int *index, *mask;
int n, size;
```

- The parameter `n` stands for the number of index elements.
- The Parameter `size` stands for the number of bytes for one element.

- For memset the target is indirectly accessed, for memget the source is indirectly addressed with indexes.

The following routines can be used if the indirect addressing is combined with a reduction operator.

```
dalib_set[m]_min_ints     (data, indexes, values, [mask,] n)
dalib_set[m]_min_reals    (data, indexes, values, [mask,]n)
dalib_set[m]_min_doubles  (data, indexes, values, [mask,]n)
dalib_set[m]_max_ints     (data, indexes, values, [mask,]n)
dalib_set[m]_max_reals    (data, indexes, values, [mask,]n)
dalib_set[m]_max_doubles  (data, indexes, values, [mask,]n)
dalib_set[m]_add_ints     (data, indexes, values, [mask,]n)
dalib_set[m]_add_reals    (data, indexes, values, [mask,]n)
dalib_set[m]_add_doubles  (data, indexes, values, [mask,]n)
dalib_set[m]_mult_ints    (data, indexes, values, [mask,]n)
dalib_set[m]_mult_reals   (data, indexes, values, [mask,]n)
dalib_set[m]_mult_doubles (data, indexes, values, [mask,]n)
dalib_set[m]_and_ints     (data, indexes, values, [mask,]n)
dalib_set[m]_or_ints      (data, indexes, values, [mask,]n)
dalib_set[m]_eor_ints     (data, indexes, values, [mask,]n)
dalib_set[m]_and_bools    (data, indexes, values, [mask,]n)
dalib_set[m]_or_bools     (data, indexes, values, [mask,]n)
dalib_set[m]_neq_bools    (data, indexes, values, [mask,]n)

int indexes[], n;

int *data, *indexes;          for dalib_set_xxx_ints
int *data, *indexes;          for dalib_set_xxx_bools
float *data, *indexes;        for dalib_set_xxx_reals
double *data, *indexes;       for dalib_set_xxx_doubles
```

The following two routines can be called with a parameter that specifies which of the above routines will be chosen. The selection is the same as the selection for the reductions (see section 12.3).

```
dalib_set[m] (int op, char *b, int *indexes, char *values,
             [int *mask,] int n);
```

13.5 Generic Memory Routines

The memory functions will be more efficient if not single bytes are transferred but full integer or double values. Therefore memory functions are implemented for integer (K=4), double (K=8) and char (K=1).

```
dalib_gather1dK (target, source, size, stride1, n1)
dalib_gather2dK (target, source, size, stride1, n1, stride2, n2)
....
dalib_gather7dK (target, source, size, stride1, n1, ..., stride7, n7)

dalib_scatter1dK (target, source, size, stride1, n1)
```

```
dalib_scatter2dK (target, source, size, stride1, n1, stride2, n2)
....
dalib_scatter7dK (target, source, size, stride1, n1, ..., stride7, n7)

dalib_memcopyK  (target, source, size)

dalib_move1dK (target, source, size, stride1, n1)
dalib_move2dK (target, source, size, stride1, n1, stride2, n2)
....
dalib_move7dK (target, source, size, stride1, n1, ..., stride7, n7)

dalib_memgetK  (target, source, index, n)
dalib_memgetpK (target, source, index, n)
dalib_memgetmK (target, source, index, mask, n)

dalib_memsetK  (target, index, source, n)
dalib_memsetpK (target, index, source, n)
dalib_memsetmK (target, index, source, mask, n)

 TYPE *target, *source;
 int index[], mask[];
 int n;

 TYPE is char for   K == 1
 TYPE is int  for   K == 4
 TYPE is double for K == 8
```

The modules *memory1*, *memory4*, and *memoryk* will be generated from the generic module *memoryK* by text replacing.

14 Sending and Receiving of Array Sections

This is now done by using derived data types.

15 Moving Distributed Array Sections

This section describes how assignments of array sections can be handled with the use of the DALIB of Adaptor.

In CMF and HPF programs there are many possibilities for moving section of distributed arrays between host and node processes.

```
      real A(N,N), B(N,N), RA(N,N)
!hpf$ distribute A  (BLOCK,BLOCK)
!hpf$ distribute B  (*,CYCLIC)
!hpf$ distribute RA (*,*)
      ...
      RA(1:N,1:N) = A(1:N,1:N)
      A(1:N,1:N)  = B(1:N,1:N)
      A(5:7,3:5)  = A(3:5,5:7)
```

Such movements will result in many point-to-point communications between the processors that have the data and the processors that need the data.

The modules `structured` and `smove` are used for the handling of such movements.

15.1 Data Movements and Schedules

Let A and B be two distributed arrays.

$$d_A : I_A \rightarrow P \qquad d_B : I_B \rightarrow P$$

Now let us consider an array assignment where values of B should replace values of A. Such a *data movement* can be described by a function m.

$$m : S_B(\subseteq I_B) \rightarrow I_A$$

For every pair (p,q) with $p \neq q$ the set $C_{p \leftarrow q}$ specifies which part of I_B processor p needs from processor q:

$$C_{p \leftarrow q} = m^{-1}(d_A^{-1}(p)) \cap d_B^{-1}(q)$$

Data movement from processor q to processor p will be necessary if $C_{p \leftarrow q}$ is not empty. Processor q has to send the corresponding data and processor p must receive it.

The collection of sets $C_{p \leftarrow q}$ is called the *schedule* of the data movement. If all sets are empty, no communication will be required and the operation is a local one.

15.2 Structured Data Movements

A data movement m will be called a *structured data movement* if every processor knows the mapping function and can compute its local schedule without any communication. This means every processor p knows the values of $C_{p \leftarrow q}$ and $C_{q \leftarrow p}$ for all processors q.

Array assignments with array sections are a typical kind of structured data movement. The following example shows the schedule that is generated for a simple array assignment when three processors are used. Only non-empty values within the schedule are of interest.

```
REAL A(1:20), B(1:13)
...
A(6:13) = B(3:10)
```

$$\begin{array}{rcl} m : \{3, ..., 10\} & \rightarrow & \{1, ..., 20\} \\ \text{i} & \mapsto & \text{i+3} \end{array}$$

$$\begin{array}{rr} d_A^{-1}(1) = \{1, ..., 6\} & d_B^{-1}(1) = \{1, ..., 4\} \\ d_A^{-1}(2) = \{7, ..., 13\} & d_B^{-1}(2) = \{5, ..., 8\} \\ d_A^{-1}(3) = \{14, ..., 20\} & d_B^{-1}(3) = \{9, ..., 13\} \end{array}$$

$$\begin{array}{rcl} m^{-1}(d_A^{-1}(1)) & = & \{3\} \\ m^{-1}(d_A^{-1}(2)) & = & \{4, ..., 10\} \\ m^{-1}(d_A^{-1}(3)) & = & \{\} \end{array}$$

As in many other situations only some values of the sets $C_{p \leftarrow q}$ are not empty.

$$C_{2 \leftarrow 1} = \{4\}, \qquad C_{2 \leftarrow 3} = \{9, 10\}$$

The following send and receive statements are generated:

```
p1: send (B(4)) to p2
    A(6) = B(7)
p2: A(8:11)=B(5:8)
    receive (A(7)) from p1
    receive (A(12:13)) from p3
p3: send (B(10)) to p2
```

15.3 Computing with Regular Sections

The module *intersection* contains functions to compute the intersection and mapping of regular sections.

These routines are used for computing local ranges of distributed arrays and for computing the schedule of moving regular sections.

```
  (l1:u1:s1)  is  section 1
  (l2:u2:s2)  is  section 2
  (l3:u3:s3)  will be the intersection of section 1 and section 2

dalib_s_s_intersect (l1, u1, l2, u2, *l3, *u3)

   - special case : s1 = s2 = 1,  note that s3 = 1

dalib_r_s_intersect (l1, u1, s1, l2, u2, *l3, *u3, *s3)

   - special case : s2 = 1   (makes computation easier)

dalib_r_r_intersect (l1, u1, s1, l2, u2, s3, *l3, *u3, *s3)
```

Here are some examples:

Figure 2: Intersection of sections.

```
dalib_r_s_intersect: (2:98:6) * (51:75) = (56:74:6)
dalib_r_s_intersect: (98:2:-6) * (51:75) = (74:56:-6)
dalib_r_s_intersect: (68:56:-6) * (26:50) = (50:56:-6)  ! is empty

dalib_r_s_intersect: (31:52:3) * (51:75) = (52:52:3)
```

While `dalib_r_r_intersect` handles all cases of intersection, the other two routines can be used if one or two strides of the input sections are equal 1.

```
mapping is defined by (s_lb: :s_str)
                   to (t_lb: :t_str)

map :   s_lb           -> t_lb
        s_lb +   s_str -> t_lb +   t_str
        s_lb + 2*s_str -> t_lb + 2*t_str
        ....

(l1:u1:s1)  is  input section
(l2:u2:s2)  is  output section, map ((l1:u1:s1))

dalib_map_section (s_lb,s_str,t_lb,t_str, l1, u1, s1, *l2, *u2, *s2)

 map : [98::-6] -> [31::4], map([98:2:-6]) = [31:95:4]
 map : [31::4] -> [98::-6], map([31:95:4]) = [98:2:-6]
 map : [98::-6] -> [31::4], map([50:2:-6]) = [63:95:4]

 map : [11::2] -> [31::3], map([11:25:2]) = [31:52:3]
```

The subroutine `dalib_map_section` is used to compute schedules for moving regular sections. Every processor has to compute which data has to be sent or to be received.

15.4 Definition of Structured Schedules

The module `smove` contains subroutines that will define structured schedules.

```
 void dalib_define_schedule (schedule_id,
                       t_top, t_dim, t_kind, t_N, t_low, t_up, t_str,
                       s_top, s_dim, s_kind, s_N, s_low, s_up, s_str)
 int *schedule_id;
 int *t_top, *t_dim, *t_kind, *t_N, *t_low, *t_up, *t_str;
 int *s_top, *s_dim, *s_kind, *s_N, *s_low, *s_up, *s_str;
```

- A mapping is defined from topology `s_top`, dimension `s_dim` to topology `t_top`, dimension `t_dim`.
- The global range `s_low:s_up:s_str` will be sent. The full size of this dimension is `s_N` and it is block or cyclic distributed (specified by `s_kind`) on the source topology.
- The global range `t_low:t_up:t_str` will be received. The full size of this dimension is `t_N` and it is block or cyclic distributed (specified by `t_kind`) on the target topology.
- The generated schedule is identified by the integer value of `schedule_id`.

```
      real, allocatable :: A(:), B(:)
!hpf$ processors PA (1:4)
!hpf$ processors PB (1:3)
!hpf$ distribute A(BLOCK) onto PA
!hpf$ distribute B(CYCLIC) onto PB
      allocate (A(N1))
      allocate (B(N2))
      ...
      A (1:K:2) = B (2*K-1:1:-2)

      parameter (K_BLOCK=1,K_CYCLIC=2)
      ...
      call dalib_define_schedule (schedule_id=IS_1,
     $     PA_TOPID, T_DIM=1, K_BLOCK, T_N=N1,
     $               T_LOW=1, T_HIGH=K, T_STR=2,
     $     PB_TOPID, S_DIM=1, K_CYCLIC, S_N=N2,
     $               S_LOW=2*K-1, S_HIGH=1, S_STR=-2)
```

In many situations, distributed dimensions are mapped to serial dimensions or serial dimensions are mapped to distributed dimensions. The following subroutines of the module `smove` can be used for these situations.

```
void dalib_define_schedule_1n_ (schedule_id,
                                          t_N, t_low, t_up, t_str,
                      s_top, s_dim, s_kind, s_N, s_low, s_up, s_str)

 void dalib_define_schedule_n1_ (schedule_id,
                      t_top, t_dim, t_kind, t_N, t_low, t_up, t_str,
                                          s_N, s_low, s_up, s_str)

 int *schedule_id;
 int *t_top, *t_dim, *t_kind, *t_N, *t_low, *t_up, *t_str;
 int *s_top, *s_dim, *s_kind, *s_N, *s_low, *s_up, *s_str;
```

The following HPF Fortran code defines movements of distributed array sections.

```
      real, allocatable :: A(:,:), B(:,:)
!hpf$ distribute A(*,BLOCK)
!hpf$ distribute B(*,CYCLIC)
      allocate (A(N,N))
      allocate (B(N,N))
      ...
      A (1:N,4) = B (6,1:N)
      A (4,1:N) = B (1:N,6)
      A (4,1:N) = B (5,1:N)
      A (1:N,4) = B (1:N,5)  ! no move definition required
```

In the generated program the following calls will be made to define schedules for these movements.

```
      call dalib_define_schedule_1n (IS_1,N,1,N,1,1,1,2,N,1,N,1)
      ...
      call dalib_define_schedule_n1 (IS_1,1,1,1,N,1,N,1,N,1,N,1)
      ...
      call dalib_define_schedule (IS_1,1,1,1,N,1,N,1,1,1,2,N,1,N,1)
      ...
```

In the next subsection it will be described how the defined schedules can be used.

15.5 Using Structured Schedules

The module *schedule* contains functions for the management of structured schedules. A structured schedule is computed for the mapping of distributed dimensions.

$$Send_p = \bigcup_{i=1}^{n} lb_i : ub_i : str_i \quad Recv_q = \bigcup_{i=1}^{n} lb_i : ub_i : str_i$$

The initialization routine `dalib_new_schedule` reserves memory on every node for a schedule that describes a movement from `send_NP` sending processors to `recv_NP` receiving processors. Structured schedules are identified by a pointer (integer value).

```
  int dalib_new_schedule (int send_NP, int recv_NP);
```

The following routines add a section to the send sets or to the receive sets.

```
  void dalib_schedule_send (int sid, int pid, int low, high, stride);
  void dalib_schedule_recv (int sid, int pid, int low, high, stride);
```

The following subroutines are later used to get access to the schedule information.

```
  void dalib_schedule_psends (int sid, int *p_low, int *p_high)
  void dalib_schedule_precvs (int sid, int *p_low, int *p_high)
```

p_low specifies the first and p_high specifies the last processor to which values have to be send or from which values have to be received.

The next two routines are used to ask for the number of sections in the union.

```
int dalib_schedule_nsends (int sid, int pid)
int dalib_schedule_nrecvs (int sid, int pid)
```

The following subroutines deliver the corresponding section that has to be send or to be received.

```
void dalib_schedule_send_range (int sid, int pid, int k,
                                int *low, int *high, int *stride, int *n)

void dalib_schedule_recv_range (int sid, int pid, int k,
                                int *low, int *high, int *stride, int *n)
```

- The input parameter `sid` identifies the schedule.
- The input parameter `pid` specifies the processor to which a section has to be sen or from it has to be received.
- The input parameter `k` is used to ask for the k-th section, $1 \leq k \leq n$, `n=schedule_nsends (sid, pid)`
- `low:high:stride` specifies the corresponding section. This section is always a subsection of the section that is owned by the calling processor.
- `n` is an additional output parameter for the number of the elements in the section.

If a distributed dimension is mapped to a serial dimension or vice versa, a processor has to send a section only to one processor or to receive a section only from one processor. In this case, it is simpler to use the following routines of the module *schedule*.

```
void dalib_schedule_send1 (int sid,
                           int *low, int *high, int *stride, int *n)

void dalib_schedule_recv1 (int sid,
                           int *low, int *high, int *stride, int *n)
```

If a schedule is no longer used, it will be released and the internal used memory is freed. This releasing is absolutely necessary for programs that define frequently different schedules.

```
void dalib_free_schedule (int sid)
```

16 Timing and Tracing

The module *timing* realizes some functions for timing. This module is very machine dependent, especially the resolution time of the timers might depend on the machine. Useful would be a resolution of at least 10 ms.

The following functions use internal timers that can be started or stopped. The current implementation realizes up to ten timers. Therefore the timer parameter must have a value between 0 and 9.

```
void dalib_clear_timer (int *timer)
void dalib_start_timer (int *timer)
void dalib_stop_timer  (int *timer)
void dalib_print_timer (int *timer)
void dalib_read_timer (int *timer, float *t)
```

The walltime subroutine returns the time in seconds since the parallel program has been started. This function is very useful for time measurements.

```
void dalib_walltime (float *t)
```

It should be observed that the timing requires a synchronization of all processors. Therefore one should be careful when calling the timer function.

The following function can be used for tracing at the moment:

```
void trace_start ();  /* called by every node */
void trace_exit ();   /* called by every node */

void trace_send (int to, int bytes);
void trace_recv (int from, int bytes);
void trace_recv_blocking (int from);
void trace_recv_waking (int from, int bytes);
```

17 Shifting of Arrays

Shifting of an array means to move the data in one dimension by some positions. If this dimension is a serial one, the shifting requires no communication. If the dimension is a distributed one, communication is required but usually only between neighbored processors.

```
void FORTRAN(dalib_cshift)

  (target_data,    source_data,    shift,     dim,
   target_section, source_section, shift_dsp, dim_dsp)

void FORTRAN(dalib_eoshift)

  (target_data,    source_data,    shift, boundary, dim,
   target_section, source_section, s_dsp, b_dsp,    d_dsp)
```

```
char         *target_data, *source_data;
section_info *target_section, *source_section;
int *dim, *shift;
char *boundary;
section_info *s_dsp, *b_dsp, *d_dsp;

   A = cshift (B, dim=1, pos = 1)
   A(1:N-1) = B(2:N); A(N) = B(1);
```

18 Indirect Addressing of Local Arrays

The module *indirect0* contains functions for indirect addressing of local arrays.

```
void dalib_local_get[m]1_ (char *a, int *a_size, int *a_elems,
                           char *b, int *b_N1, int p1[], int mask[])

void dalib_local_get[m]2_ (char *a, int *a_size, int *a_elems,
                           char *b, int *b_N1, int *b_N2,
                                    int p1[], int p2[], int mask[])

void dalib_local_get[m]3_ (char *a, int *a_size, int *a_elems,
                           char *b, int *b_N1, int *b_N2, int *b_N3,
                           int p1[], int p2[], int p3[], int mask[] )

void dalib_local_get[m]4_ (char *a, int *a_size, int *a_elems, char *b,
                           int *b_N1, int *b_N2, int *b_N3, int *b_N4,
                           int p1[], int p2[], int p3[], int p4[], int mask[])
```

- The parameter `a` is a pointer to the first element of the target array. `a_elems` values will be accessed, every value has `a_size` elements.
- The parameter `b` is the source array that is indirectly addressed. Its rank is k and the extension of the dimensions are `b_N1`, `b_N2`, ..., `b_Nk`.
- The integer arrays `p1`, `p2`, ..., `pk` are the indexes. They have also `a_elems` values.

```
void dalib_local_setm1_ (int *op, char *b, int *b_N1, int p1[],
                         char *a, int *a_size, int *a_elems, int mask[])

void dalib_local_setm2_ (int *op, char *b, int *b_N1, *b_N2,
                         int p1[], int p2[],
                         char *a, int *a_size, int *a_elems, int mask[])

void dalib_local_setm3_ (int *op, char *b, int *b_N1, *b_N2, *b_N3,
                         int p1[], int p2[], int p3[],
                         char *a, int *a_size, int *a_elems, int mask[])

void dalib_local_setm4_ (int *op, char *b, int *b_N1, *b_N2, *b_N3, *b_N4,
                         int p1[], int p2[], int p3[], int p4[],
                         char *a, int *a_size, int *a_elems, int mask[])
```

- The parameter `a` is a pointer to the first element of the source array. `a_elems` values will be put in the target array, every value has `a_size` elements.
- The parameter `b` is the target array that is indirectly addressed. Its rank is k and the extension of the dimensions are `b_N1`, `b_N2`, ..., `b_Nk`.
- The integer arrays `p1`, `p2`, ..., `pk` are the indexes. They have also `a_elems` values.
- The parameter `op` describes the operation on the destination array. Values can be copied, but it is also possible to build sums, to find maxima or minima.

```
parameter (M1=m1,...,Mk=mk,N=n)
TYPE*b B(M1,...Mk), A(N)
INTEGER P1(N), ..., Pk(N)
LOGICAL MASK (N)

FORALL (I=1:N)
   A(I) = B(P1(I),...,Pk(I))
END FORALL
```

This loop with indirect addressing on serial arrays is exactly realized with the following subroutine call:

```
      call dalib_local_getmk (A, b, N, B, M1, ..., Mk,
     $                                   P1, ..., Pk, MASK)

      DO I = 1, N
         B(P1(I),...,Pk(I)) = B(P1(I),...,Pk(I)) + A
      END DO

      call dalib_local_setmk (17, B, M1, ..., Mk,
                              P1, ..., Pk, A, b, N, MASK)
```

The above subroutines can be used with any contiguous array section instead of a one-dimensional array A.

In the module *indirect0* there are procedures being realized for calculating the indexes. These procedures will also be used for the indirect addressing of distributed arrays.

```
void dalib_get_[m]pointersk (no_ptrs, target_pointers, [mask,]
                             N1, index1, ..., Nk, indexk)
```

This subroutines builds single value addresses for an indirectly addressed array.

```
target_pointers[i] =
    ((...indexk[i] * N<k-1) + index<k-1> ...) * N1 + index1
```

19 Indirect Addressing of Distributed Arrays

In this section all modules are discussed that can be used to realize indirect addressing for distributed arrays.

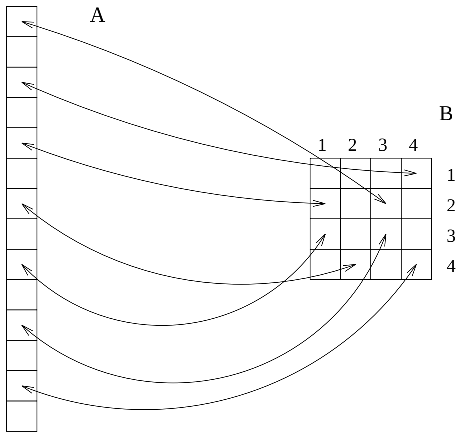

Figure 3: Indirect addressing of a topology

19.1 Notations and Example

The following code shows a typical example for indirect addressing. There are two kinds of operations: one is the gather operation (indirect read) and one is the scatter operation (indirect write).

```
      TYPE*b B(4,4), A(14)
      INTEGER P1(14), P2(14)
      LOGICAL MASK (14)
!HPF$ DISTRIBUTE A (...)
!HPF$ DISTRIBUTE B (...,...)
!HPF$ ALIGN WITH A :: P1, P2, MASK
      ...
      FORALL (I=1:14)
         A(I) = B(P1(I),P2(I))     ! gather operation
         B(P1(I),P2(I)) = A(I)     ! scatter operation
      END FORALL
```

Figure 3 shows a mapping between two arrays. One array is the source array (A) and one array is the target array (B).

The mapping is always defined by n integer arrays where n is the rank of the target array. The size, shape and distribution of these integer arrays must be the same as the source array. A mask array can also be used to restrict the mapping to the relevant elements of the source topology.

Note: The mapping between two topologies is always defined by integer arrays and a mask that are conform to the source array. Two elements of the source can have the same target (no injective mapping).

```
        | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
  ------|----|---|---|---|---|---|---|---|---|---|---|---|---|---|-
   MASK | T  | F | T | F | T | F | T | F | T | F | T | F | T | F |
    P1  | 2  | - | 2 | - | 1 | - | 4 | - | 3 | - | 3 | - | 4 | - |
    P2  | 3  | - | 4 | - | 1 | - | 2 | - | 1 | - | 3 | - | 4 | - |
```

A gather or a scatter operation is an *unstructured data movement* where communication is required to compute the schedule. Such a data movement has two steps of communication.

The modules `unstructured` and `indirect` realize functions for indirect addressing of distributed arrays. The central idea is that different processors can access data of other processors or send them data, where the communication pattern itself will be known only at run-time.

19.2 Handling of global Pointers

The first step of computing the schedule is to find out on which processors the indexed elements are located and to translate the global addresses of the target array B to local addresses.

```
      TYPE B (N1, ..., Nk)
!HPF$ DISTRIBUTE B (kind1, ..., kindk) ONTO <topid>
```

Every processor computes for its local part of the source array and so for its local part of indexes and mask the corresponding target processors.

```
void dalib_get_[m]targets (no_ptrs, target_procs, [mask,] topid,
                           N1, kind1, index1,
                           ....
                           Nk, kindk, indexk)

    int no_ptrs;          // is number of indexes

    int **target_procs;   // returns array of target processor ids

    int topid;            // specifies topology, rank must be k

    int N1, ..., Nk;      // specifies size of dimensions
    int kind1, .., kindk; // specifies kind of distribution

    int mask[no_ptrs];    // specifies a mask for computing (optional)

    int index1[no_ptrs],
        ....
        indexk[no_ptrs];  // index values for the dimensions
```

For the example shown in Figure 3 this routine will give the following results if a two-dimensional block distribution of the array B on 2×2 processors is assumed.

```
 call dalib_get_mtargets (n, procs[], mask[], B_topid,
                          4, BLOCK_KIND, index1[],
                          4, BLOCK_KIND, index2[])

      | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
------|----|---|---|---|---|---|---|---|---|---|---|---|---|---|-
 mask | T  | F | T | F | T | F | T | F | T | F | T | F | T | F |
index1 | 2  | - | 2 | - | 1 | - | 4 | - | 3 | - | 3 | - | 4 | - |
index2 | 3  | - | 4 | - | 1 | - | 2 | - | 1 | - | 3 | - | 4 | - |

 procs | 3       3       1       3       3       4       4
```

Furthermore, the following routine is used to translate the global addresses to local addresses. By this way every processor of the target topology can easily identify the position of the requested data.

```
void dalib_update_target_addressesk (int target_addresses[],
                                     int nr, int topid, int s_size,
                                     int N1, kind1,
                                     ...
                                     int Nk, kindk)

  int target_addresses[];   // input : global , output : local addresses
  int nr;                   // size of array target_addresses

  int topid;                // abstract processor array for target array
  int N1, ..., Nk;          // size of target array
  int kind1, ..., kindk;    // distribution of different dimensions

  int s_size;               // size of one element in target_array
```

19.3 Schedules for unstructured movement

Every processor p can compute the values $C_{p \leftarrow q}$, that is the data required from other processors. This is possible as it is assumed that the index arrays, the mask array and the source array have all the same shape and distribution.

But a processor does not know the values $C_{q \leftarrow p}$, that is the data needed by other processors. Therefore every processor p computes $C_{p \leftarrow q}$ and sends a description of this set to the processor q to inform it about the needed values. The corresponding schedule is computed explicitly in a corresponding data structure.

An unstructured schedule can be generated by the following subroutine of the module `unstructured`:

```
void dalib_new_indirect (schedule_ptr *new_schedule,
                         int source_topid, int target_topid,
                         int N, target_proc[], target_indexes[])

   source_topid  : identifier for topology of source
   target_topid  : identifier for topology of target

   N           : number of accesses from source to target

   target_proc [i]    : target processor for i-th element
                       (<= 0 implies no access)

   target_indexes [i] : target address for i-th element

   new_schedule: output is a pointer to the unstructured comm. schedule
```

A mapping is defined from the topology `source_topid` to the topology `target_topid`.

The parameters `N`, `translation` and `target_indexes` describe the values $C_{p \leftarrow q}$ for the calling process p.

- The parameter `N` is the total number of entries.
- `target_indexes[i]` specifies for $0 <= i < N$ the global index of the globally addressed structure.
- `target_proc[i]` specifies for $0 <= i < N$ the processor from which the data is needed. Therefore this subroutine does not have to compute where the global data resides.

With the given information the processors exchange the information which data is needed. Afterwards every processor has the values $C_{q \leftarrow p}$, that is the data needed by other processors.

```
typedef struct indirect_schedule *schedule_ptr;

struct indirect_schedule {

  int source_topid;
  int target_topid;

  int no_source_indexes;   /* number of elements of source indexes */

  int *source_indexes;     /* pointers from the source array
                              size of array is no_source_indexes   */

  int no_target_indexes;   /* indexes accessed by other processors */

  int *target_indexes;     /* pointers into the target topology    */

  dd_type *source_ddt;     /* descriptor for sending from source   */
  dd_type *target_ddt;     /* descriptor for receiving in target   */
};
```

The following example demonstrates the functionality of the routines of this module.

Every processor has its information about the global indexes and on which processors they are.

```
                       processor 1          processor 2
                       ===========          ===========

target_addresses:     <2, 5, 8, 7, 8>      <10, 9, 6, 2, 6>
target_proc:          <1, 1, 2, 2, 2>      < 2, 2, 2, 1, 2>
```

Now the schedule for the unstructured movement can be generated.

```
dalib_new_indirect (&s_ptr, 1, 1, 10, translation[], target_indexes[])
```

The processors exchange the information how many data is needed on other processors. If $n_{p \leftarrow q}$ is the number of elements that processor p needs from q, this number will be sent from p to q and received by q from q.

```
              | needed from    |    needed from
              | processor 1    |    processor 2
--------------|----------------|------------------
 processor 1  |      2         |        3
 processor 2  |      1         |        4
```

The target_addresses are also exchanged that every processor knows about the values needed by other processors.

```
              | needed from    |    needed from
              | processor 1    |    processor 2
--------------|----------------|------------------
 processor 1  |    < 2, 5 >    |   < 8, 7, 8 >
 processor 2  |     < 2 >      |   < 10, 9, 6, 6 >
```

These values are stored in the corresponding data structure for the schedule.

```
                  processor 1       processor 2
                  ===========       ===========

no_target_indexes      3                  7
target_indexes       <2, 5 | 2>      <8, 7, 8 | 10, 9, 6, 6>
```

The values of `target_no` and `target_ptr` are needed to find out to which sources the target addresses belong.

When the processors get their requested values it is necessary to know to which position this belongs. The source indexes indicates on which position the received data must be put.

```
              | positions for  |    positions for
              | processor 1    |    processor 2
--------------|----------------|------------------
 processor 1  |    < 1, 2 >    |   < 3, 4, 5 >
 processor 2  |    < 4 >       |   < 1, 2, 3, 5>

                  processor 1         processor 2
                  ===========         ===========
no_source_indexes      5                    5
source_indexes    <1, 2 | 3, 4, 5>    <4 | 1, 2, 3, 5>
```

The information about an unstructured schedule can be retrieved with the following subroutines:

```
void dalib_indirect_source_info (int s_id, int *no_values,
                                 int **indexes)

void dalib_indirect_target_info (int s_id, int *no_values,
                                 int **indexes)
```

For sending and receiving data the following subroutines are used.

```
  void dalib_indirect_send_target (schedule_ptr *sptr, char *values, int size)
  void dalib_indirect_recv_target (schedule_ptr *sptr, char *values, int size)
```

Here the value of `target_no[j]` specifies how many values has to be sent or to be received from processor j.

```
  void dalib_indirect_send_source (int nr, char *values, int size)
  void dalib_indirect_recv_source (int nr, char *values, int size)
```

Here the value of `source_no[j]` specifies how many values has to be sent or to be received.

```
  void dalib_free_indirect (schedule_ptr sptr)
```

19.4 Realization of Global Get and Global Set

The module *indirect* contains functions that make the realization of indirect addressing possible.

```
  dalib_ind_definemk_  (s_id, A_topid, A_N, mask, B_topid,
                        N1, kind1, index1,
                        ....
                        Nk, kindk, indexk)

  dalib_ind_definek_  (s_id, A_topid, A_N, B_topid,
                       N1, kind1, index1,
                       ....
                       Nk, kindk, indexk)

  void dalib_ind_get_send_ (*s_id, char *data, int *bytes)
  void dalib_ind_get_recv_ (*s_id, char *data, int *bytes)

  void dalib_ind_set_send_ (*s_id, char *data, int *bytes)
  void dalib_ind_set_recv_ (*s_id, int *op, char *data, int *bytes)

  void dalib_ind_free_ (int *s_id)

void dalib_ind_get_send_ (s_id, b, b_size)

int *s_id, *b_size;
unsigned char *b;

{ unsigned char *values;
  int size, no_values;
  int *local_indexes;

  size = *b_size;

  dalib_indirect_target_info (*s_id, &no_values, &local_indexes);

  /* Step 1: use schedule to get values needed by other processes */
```

```
  values = (unsigned char *) dalib_malloc (no_values * size, "ind_get_send");

  dalib_memget (values, b, local_indexes, no_values, size);

  /* Step 2: send these values to other processors */

  dalib_indirect_send_target (*s_id, values, size);

  free (values);
} /* dalib_ind_get_send_ */
```

19.5 Translation with Adaptor

```
      integer A(N), B(N), P(N)
!hpf$ distribute (*) :: HA
!hpf$ distribute (CYCLIC) :: A
!hpf$ align with A :: B, P
      ...
      call global_get (B, A, P)
!     or
      B = A(P)

      call dalib_ind_define1 (IS_1,1,B_SIZE1,1,N,1,P)
      call dalib_ind_get_send (IS_1,A,4)
      call dalib_ind_get_recv (IS_1,B,4)
      call dalib_ind_free (IS_1)

      B(P) = A
!     or
      call global_send (B, P, A)

      call dalib_ind_define1 (IS_1,1,A_SIZE1,1,N,1,P)
      call dalib_ind_set_send (IS_1,A,4)
      call dalib_ind_set_recv (IS_1,0,B,4)
      call dalib_ind_free (IS_1)
```

For the global send operation a mask can be used. It is also possible to specify what should happen if more than one value is send to the same position.

The operation op has the same meaning as in the reductions. The value 0 specifies that the send values are copied. Therefore it might be possible that there will different results when indexes appear twice.

When using global get and send the following restrictions have to be observed (e.g. this is checked within the Adaptor translation system):

- all of the arrays must be distributed
- The index arrays p1, ..., pk must be integer*4 arrays
- a and b must have the same type, no implicit type conversion is done
- a and the index arrays p1, ..., pk must have the same shape and the same distribution (at the moment only block distributions are possible).

19.6 Indirect Addressing with Overlapping

```
      PARAMETER (N1=10000,N2=12000)
      REAL DIREC1(N2), DIREC2(N1)
      INTEGER LCC(6,N1)
      ...
      DO NC = 1, N1
        DIREC2(NC)=BP(NC)*DIREC1(NC)
     X            -BS(NC)*DIREC1(LCC(1,NC))
     X            -BW(NC)*DIREC1(LCC(4,NC))
     X            -BL(NC)*DIREC1(LCC(5,NC))
     X            -BN(NC)*DIREC1(LCC(3,NC))
     X            -BE(NC)*DIREC1(LCC(2,NC))
     X            -BH(NC)*DIREC1(LCC(6,NC))
      END DO
      ...
      DIREC1(1:N1) = DIREC2(1:N1)

      REAL DIREC1(N1), DIREC2(N1)

      DO NC = DIREC2_LOW, DIREC2_HIGH
        DIREC2(NC)=BP(NC)*DIREC1(NC)
     X            -BS(NC)*DIREC1(LCC(1,NC))
     X            -BW(NC)*DIREC1(LCC(4,NC))
     X            -BL(NC)*DIREC1(LCC(5,NC))
     X            -BN(NC)*DIREC1(LCC(3,NC))
     X            -BE(NC)*DIREC1(LCC(2,NC))
     X            -BH(NC)*DIREC1(LCC(6,NC))
      END DO

      LCC(J,NC) > N1 : index put in boundary section of DIREC1
      LCC(J,NC) > DIREC2_HIGH or
      LCC(J,NC) < DIREC2_LOW  is a non local index

      call dalib_ind_overlap (IS_1,1,6*LCC_SIZE1,N1,1,LCC)
      DIREC1(DIREC1_HIGH1+1) = 0.0
      call dalib_ind_get_send (IS_1,DIREC1,4)
      call dalib_ind_get_ov_recv (IS_1,DIREC1(DIREC1_HIGH1+1),4)
      call dalib_ind_free (IS_1)
```

20 Shared Arrays

20.1 Operations for Shared Arrays

Shared arrays can be accessed (read or write) by all processors by one-sided communication. An array has to be defined as shared before it is allocated.

```
FUNCTION(dalib_array_shared) (array_info array_id, int *kind)
```

Internally, every shared array is described by a record that indicates how the array is shared.

```
struct SharedRecord

 { int   shared_kind;
   ...
 };
```

If an array is shared, every value can be read or written by every processor.

```
    FUNCTION(dalib_array_read) (char *data, array_info *array_id,
                                int *ind1, int *ind2, ..., int *ind7)

    FUNCTION(dalib_array_write) (char *data, array_info *array_id,
                                 int *ind1, int *ind2, ..., int *ind7)
```

As the single element access requires too much overhead the DALIB provides also routines to access many values (like indirect addressing).

```
      FUNCTION(dalib_array_remote) (remote_ptr *s_id,
                         char *array_vals, array_info *array_dsp
                         char *base_vals, array_info *base_dsp,
                         char *mask_vals, array_info *mask_dsp,
                         char *ind1_vals, array_info *ind1_dsp,
                         ...
                         char *ind7_vals, array_info *ind7_dsp)

      FUNCTION(dalib_array_rem_read) (remote_ptr *s_id,
                                      array_info *array_dsp,
                                      array_info *base_dsp )

      FUNCTION(dalib_array_rem_write) (remote_ptr *s_id,
                                       array_info *array_dsp,
                                       array_info *base_dsp )

      FUNCTION (dalib_remote_free) (remote_ptr *s_id)


      int dalib_is_globally_shared (array_info array_id)
end{verbatim}
}
{\scriptsize
\begin{verbatim}
      void dalib_array_shared_alloc (array_info array_id)


    void dalib_shared_free (array_info array_id, int data_flag)


#define MEM_SHM_SHARED       1
#define MEM_RPC_SHARED       2
/* define SHM  : module shm is available   */
/* define RPC  : module rpc is available   */
```

20.2 Remote Offsets

Usually, an array is indexed by up to 7 dimensions. Every processor can compute the offset on other processors.

20.3 Shared Memory Segments

```
dalib_shm_allocate (shm_size => char *shm_ptr, int shm_id)

dalib_shm_deallocate (char *shm_ptr, int shm_id)
```

Internally, one processor creates the shared memory segment while all the others attach to it.

```
if (pcb.i == 1)
   shm_id = shm_allocate (shm_size);
dalib_broadcast (shm_id, sizeof(int), 1);
shm_attach (shm_id)
```

20.4 Remote Procedure Calls

The following routine starts on a processor the possibility that it can be interrupted by other proccessors.

```
void rpc_init ()
```

As there might be conflicts with the message passing system used in the DALIB the server mode can be determined explicitly.

```
void rpc_exit ()
```

```
void rpc_call (int pid, char *query, int query_size,
                        char *answer, int answer_size)
```

This routine will invoke the following routine on the processor pid:

```
void dalib_answer_question (char *query, int query_size,
                            char *answer, int answer_size)
```

The result is given back to the calling processor.

The data structures for queries and answers can be defined in any way.

```
typedef struct {

  int kind;              /* Attention: must not be 0 (internally used) */
  array_info *array_id;
  int offset;

  } question_type;
```

The described mechanism is used to implement the following two routines:

```
void dalib_rpc_get (data, remote_pid, array_id, offset)
void dalib_rpc_put (data, remote_pid, array_id, offset)
```

21 Intrinsics and HPF Library

21.1 Random Numbers

The module 'random' realizes functions to get random numbers. The functions can be used for any kind of sections of arrays. Internally used is a parallel random number generator with a very high period.

```
/* initializing for new sequences based on the given seed */
void dalib_random_init (seed)
int *seed;

/* random numbers (integer*4) in range 0 through limit - 1 */

void dalib_section_int_randoms (section_info *section_id, int *limit);

/* random numbers (single or double) in range 0.0 through 1.0 */

void dalib_section_real_randoms (section_info *section_id, int *limit)
```

Within Adaptor the functions are used to realize the Fortran 90 intrinsic routines for random numbers, namely RANDOM_NUMBER and RANDOM_SEED.

```
real A (M, N)      ! A is distributed
call random_data (A)
```

Adaptor generates the following code:

```
call dalib_section_real_randoms (A_DSP)
```

As the descriptor of a section or an array contains also the size information for one element, the same routine is used for getting single or double real values.

The module for random values can recognize automatically whether the target hardware uses litle endian or big endian data format. Nevertheless, the random routines for the initialization of the parallel random number generator can be machine-dependent.

21.2 Timing

21.3 Sorting

21.4 Prefix and Suffix Operations

Acknowledgements

The most thanks are due to David Greco, CRS4, Sardenia (Italy), for many fruitful discussions about the design of descriptors for arrays and distributions.

References

[Bra96] T. Brandes. ADAPTOR Programmer's Guide(Version 4.0). Technical documentation, GMD, April 1996. Available via anonymous ftp from `ftp.gmd.de` as `gmd/adaptor/docs/pguide.ps`.

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

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

[Mes94] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Draft Version February 1994 1.0, University of Tennessee, Knoxville, February 1994.