Contents

ADAPTOR (Version 1.0)
The Distributed Array Library

T. Brandes

Internal Report No. Adaptor 4

April 6, 1993

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

ADAPTOR (Version 1.0)
The Distributed Array Library

T. Brandes

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

Abstract

ADAPTOR (Automatic Data Parallelism Translator) is a tool that transforms data parallel programs written in Fortran with array extensions, parallel loops, and layout directives to parallel programs with explicit message passing.

ADAPTOR is not a compiler but a source-to-source transformation. The generated sources are Fortran programs that call functions of a library, that realizes functions for communication and global operations on distributed arrays. This library is called DALIB (distributed array library) and is described in this paper.

1 Introduction

For the realization of the communications needed for global operations on distributed arrays, a basic set of library functions will be used that build the DALIB (distributed array library).

- low level communication (send, receive, wait, ...),
- high level communication (broadcast, reduction, barrier, ...),
- generic functions for typed communication,
- functions to control the data partitioning,
- functions to move and replicate sections of distributed arrays,
- timing and tracing facilities,
- parallel random number generator,
- X-Windows interface.

The DALIB is implemented in C. Though most part of this library is portable, the high level communication routines should be optimized for the underlying hardware architecture.

The following is a short overview of all modules that realize the distributed array library.

```
system.h          ! general include file
timing.c          ! timing functions
timing1.c         ! timing functions, only single node
random.c          ! random numbers
mailbox.c         ! primitive send and recv with mailboxes
barrier.c         ! barrier synchronization
broadcast.c       ! broadcast from one node to all other nodes
reduction.c       ! reduction functions (sum, min, max, ...)
buffer.c          ! filling and sending communication buffers
memcopy.c         ! efficient memory copy
manager.c         ! management of array distributions
section.c         ! help functions for send/recv of sections
hostnode.c        ! transfer of arrays between host and node
replicate.c       ! replication of distributed data
cshift.c          ! circular shifting of distributed arrays
movement.c        ! sending and receiving of array sections
schedule.c        ! indirect addressing
trace.c           ! trace utilities

mhost.c           ! main program of host process
mnode.c           ! main program of node process
mcube.c           ! main program of node process (without host)
```

The library has been realized for different versions.

- One version is based on PVM. At the moment PVM version 2.4 and version 3.0 can be used. This version should run on all machines where PVM is running.
- One version is based on communication via a shared memory segment. This version has been realized for the Alliant FX/2800, the KSR 1 and Silicon Graphics machines.
- For the CM 5, GC and the iPSC/860 the library has been realized based upon the native communication routines.

2 Main Programs

Adaptor knows about three different kinds of processes for the generated parallel program:

- **host** is the process that is called on the front end of the parallel machine; it starts the node processes and executes itself the generated host program of ADAPTOR
- **node** is the process that runs on all nodes of the parallel machine, these processes are usually created by the host process and run all the generated node program of ADAPTOR
- **cube** is the process that runs on all nodes of the parallel machine without a host process, these processes have usually to be loaded on all nodes and run all the generated program of ADAPTOR if the only node mode has been chosen.

For every process there is an own main program. For some machines it might be possible that there is the same main program for host and node, because the host process forks itself (Alliant FX/2800) or one node of the parallel machine is executing the host program (Parsytec GC). In this case the id of the process is used to find out whether the host program or node program has to be called.

```
int main()
   - inits random area, initializes mailboxes (mailbox)
   - creates node processes (if host)
   - starts function hostmodule (host) or nodemodule (node, cube)
   - waits for termination of all node processes
```

The function hostmodule and nodemodule are generated by ADAPTOR from the original source program.

```
PROGRAM TEST    ! source program for ADAPTOR
....

SUBROUTINE HOSTMODULE ()    ! in generated source: host.f
....

SUBROUTINE NODEMODULE ()    ! in generated source: node.f
....
```

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

```
int dalib_pid       ()
    - returns id of calling process

int dalib_nproc ()
    - returns number of node processes
```

The host process has always the id 0, the node processes are numbered from 1 to p where p is the number of created node processes.

3 Communication

3.1 Module 'mailbox'

This module realizes all message passing between the processes. 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_receive (from, message, length)
void dalib_send    (to,   message, length)

int *from, *to;
char *message;
int *length;
```

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 following properties are very important:

- asynchronous message passing
- 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.

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)

3.2 Module 'barrier'

A barrier realizes a synchronization between all processes. The corresponding function has to be called by all processors.

```
   void process_barrier ()  /* synchronization of all node processes */
```

3.3 Module 'broadcast'

With broadcast values of nodes and host can be exchanged.

```
dalib_broadcast (data, size, j)
  - broadcast to host and all node from process j

dalib_node_broadcast (data, size, j)
  - broadcast to host and all node from process j

char *data;
int *j, *size;       /* process j sends the data */
```

3.4 Module 'reduction'

This module realizes different reduction operations for all node processors.

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 (dat, op)
unsigned 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 maximmum value appears.

```
void dalib_reduction (dat, op)
unsigned 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 mimimum or maximum value.

```
void dalib_loc_exchange (data, size)
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(100), 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   ! A is not in host   real S, A(:)
...                           ...
call dalib_reduction (S,8)    S = sum (A)      ! local sum on own part
                              call dalib_reduction (S,8)
call dalib_pos_reduction (M,4) M = minval (A)
                              call dalib_pos_reduction (M,4)
                              P = minloc (A)
call dalib_loc_exchange (P,4)  call dalib_loc_exchange (P,4)
```

3.5 Module 'buffer'

Sometimes it might be useful to send and receive data that does not belong to a continguous section. In this case the data should be copied to a buffer.

For this purpose a special module has been realized.

```
Function    : Combining of messages in a buffer (send + recv)

is used for combining non-continguous sections to one message
only for internal use in the library)

Export :

  Filling and sending the buffer

     dalib_create_buffer (size, 0);
     dalib_fill_buffer (ptr1, size1);
                ...
     dalib_fill_buffer (ptrn, sizen);
     dalib_send_buffer (to);
     dalib_destroy_buffer ();

  Receiving and getting from the buffer

     dalib_create_buffer (size, 1);
     dalib_recv_buffer (from);
     dalib_get_buffer (ptr1, size1);
               ...
     dalib_get_buffer (ptrn, sizen);
     dalib_destroy_buffer ();
```

4 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.

```
void dalib_clear_timer (timer)
void dalib_start_timer (timer)
void dalib_stop_timer  (timer)
void dalib_print_timer (timer)
int *timer;     /* 0 <= *timer < TIMERS (=10)  */
```

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 (t)
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 (to, bytes)
int to, bytes;

void trace_recv (from, bytes)
int from, bytes;

void trace_recv_blocking (from)
int from;

void trace_recv_waking (from, bytes)
int from, bytes;
```

5 Distribution of Arrays

The module 'manager.c' is very important as here the internal information about all distributions is managed. The functions are needed to determine which elements the processes have and what the local and global shape of distributed variables are.

The following has to be observed when using these functions:

- For the curren version only one dimension can be used for the distribution of the array.
- The distributed dimension is always the last dimension.

5.1 Mapping a Distributed Array to Processes

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.

Let A be a onedimensional 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)
```

If A has more than one dimension, only the last index is used for the mapping.

5.2 Definition of Array Descriptors

Every distributed array has to be defined by the host process and by all node processes. Afterwards it is possible to get information about the layout of the distributed array, e.g. which element is owned by which processor.

```
void dalib_define_array1  (a_dsp, size, lb1, ub1)
void dalib_define_array2  (a_dsp, size, lb1, ub1, lb2, ub2)
void dalib_define_array3  (a_dsp, size, lb1, ..., lb3, ub3)
void dalib_define_array4  (a_dsp, size, lb1, ..., lb4, ub4)

int *a_dsp, *size, *lb1, *ub1, ..., *lb4, *ub4;

  - size : number of bytes for one element
  - [lb1:ub1,lb2:ub2] is shape of the distributed array
  - a_dsp is output value that has to be used in other oper

void dalib_undefine_array (a_dsp)
```

- Every distributed array has to be defined with the corresponding subroutine call.
- The calls of define and undefine have to be exactly in the opposite order.

The definition of array descriptors is used within ADAPTOR in the following way:

```
real A(N)

integer*4 A_DSP
real A(:)       ! only in node program
call dalib_define_array1 (A_DSP,4,1,N)
....
call dalib_undefine_array (A_DSP)
```

5.3 Local and Global Shape of Distributed Arrays

After a distributed array has been defined, one can use some functions to get information about the local and/or global shape.

```
void dalib_array_dimensions1 (a_dsp, n1, n2)
   - [n1:n2] is shape of local part of array a_dsp

void dalib_array_dimensions2 (a_dsp, nl1, nr1, nl2, nr2)
   - [ nl1:nr1, nl2:nr2 ] is the shape of the array
```

```
void dalib_array_dimensions3 (a_dsp, nl1, nr1, nl2, nr2, nl3, nr3)
   - [ nl1:nr1, nl2:nr2 , nl3:nr3] is the shape

void dalib_array_dimensions4 (a_dsp, nl1, nr1, ..., nl4, nr4)
   - [ nl1:nr1, nl2:nr2 , nl3:nr3, nl4:nr4] is the shape

void dalib_array_pardim (a_dsp, n1, n2)
   - [n1:n2] is local extension of the parallel dimension

void dalib_pardim_dimensions (a_dsp, n1, n2)
   - [n1:n2] is global extension of the parallel dimension
```

- The first subroutines are used to get information about the local extensions of the distributed array.
- The last subroutine is used to get information about the global extensions of the distributed dimension.

```
real A(N)

integer*4 A_DSP
integer*4 A_LOW, A_HIGH
real A(:)
call dalib_define_array1 (A_DSP,4,1,N)
call dalib_array_pardim (A_DSP,A_LOW,A_HIGH)
ALLOCATE (A(A_LOW:A_HIGH))
....
call dalib_undefine_array (A_DSP)
```

The following subroutines are used to get information about the size (in bytes) of a local part of a distributed array. These subroutines are also used to get this information about the local part of other processes.

```
void dalib_array_size  (a_dsp, i, size)
int *a_dsp, *i, *size;
/* size is the number of bytes that process i needs for
   the distributed array specified by a_dsp              */
```

5.4 Slices and Sections of Distributed Arrays

When an array is accessed with a range, every process has to determine the local range. This is only necessary for the distributed dimension.

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

void dalib_pardim_range_ (a_dsp, global_lb, global_ub, global_stride,
                                 local_lb, local_ub, local_stride)
int * a_dsp;
int * global_lb, * global_ub, * global_stride;
int * local_lb, * local_ub, * local_stride;

void dalib_pardim_slice_ (a_dsp, global_lb, global_ub,
                                 local_lb, local_ub   )
int * a_dsp;
int * global_lb, * global_ub;
int * local_lb, * local_ub;
```

```
      real a(1000)
      ...
      a(1:1000)    = 0.0
      a(4:996)     = 1.0
      a(10:990:10) = a(10:990:10) * 5.0
      end
```

This will be generated for the node programs:

```
      INTEGER*4 A_DSP
      REAL*4 ADP_NODE_GET_DA_R4_1
      INTEGER*4 A_LOW, A_HIGH
      INTEGER*4 A_START, A_STOP, A_INC
      REAL*4 A (:)
      call dalib_define_array1 (A_DSP,4,1,1000)
      call dalib_array_pardim (A_DSP,A_LOW,A_HIGH)
      ALLOCATE (A(A_LOW:A_HIGH))
      A(A_LOW:A_HIGH) = 0.0
      call dalib_pardim_slice (A_DSP,4,996,A_START,A_STOP)
      A(A_START:A_STOP) = 1.0
      call dalib_pardim_range (A_DSP,10,990,10,A_START,A_STOP,A_INC)
      A(A_START:A_STOP:A_INC) = A(A_START:A_STOP:A_INC)*5.0
      call dalib_undefine_array (A_DSP)
      DEALLOCATE (A)
```

5.5 Exchanging of single elements

The following subroutine will be called to exchange a single element of a distributed array. The owner process copies the node data to the replicated scalar and broadcasts its value.

```
void dalib_node_get_ (rep_data, node_data, a_dsp, index)
int *a_dsp;
int *index;
unsigned char *rep_data, *node_data;
```

The following example shows the typical use within Adaptor:

```
      real a, b(100,50)
cmf$  layout b(:news))
      a = b(i,j)

c     host program
      REAL A
      call dalib_node_get (a, a, b_dsp, j)

c     node program
      REAL A, B(:)
      ...
      call dalib_node_get (a, b(i,j), b_dsp, j)
```

The next function is a logical function that returns true if the calling processor owns the element or column of the distributed data structure.

```
int dalib_have_i (a_dsp, index)
int *a_dsp;
int *index;
```

As other processes have to know sometimes which process owns a certain element of a distributed array, the following function becomes necessary:

```
int dalib_who_has (a_dsp, k)
int *a_dsp, *k;
/* returns id of the process that owns k-th column */
```

6 Random Numbers

The module 'random' realizes functions to get random numbers but always random numbers for the whole distributed array. 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 ub-1 */
void dalib_get_int_randoms (a_dsp, a_data, ub)
int *a_dsp, *ub;
int a_data[];

/* random numbers (single precision) in range 0.0 through 1.0 */
void dalib_get_real_randoms (a_dsp, a_data)
int *a_dsp;
float a_data[];

/* random numbers (double precision) in range 0.0 through 1.0 */
void dalib_get_double_randoms (a_dsp, a_data)
int *a_dsp;
double a_data[];
```

7 Moving of Sections of Distributed Arrays

In Fortran 90 programs supported by ADAPTOR there are many possibilities for moving section of distributed arrays between host and node processes.

```
      real A(N,N), RA(N,N), HA(N,N)
cmf$  layout RA(:serial), HA(:host)
      ...
      RA(1:N,1:N) = A(1:N,1:N)
      HA(1:N,1:N) = A(1:N,1:N)
      A(1:N,1:N)  = HA(1:N,1:N)
      A(5:7,3:5)  = A(3:5,5:7)
```

The current implementation for moving sections of distributed arrays has the following restrictions:

- sections with strides not equal 1 cannot be handled,
- the sections of host arrays or replicated arrays must be continguous, but not the ones of the distributed arrays.

7.1 Local and Global Moving of Data

The module 'memcopy' contains very fast functions for copying continguous memory sections. The copy function check how the data is aligned to use the fastest copy possibility.

```
void dalib_memcpy (target, source, size)
unsigned char *target, *source;
int size;

void dalib_rmemcpy (target, source, size)
unsigned char *target, *source;
int size;
```

These functions are also used to fill a buffer (see section 3.5) for sending and receiving non-continguous sections.

7.2 Module 'section'

The idea of this module is to define a section of a distributed array and to make operations with this local section (sending, receiving, copying).

```
void dalib_setup_array (a_dsp)
int a_dsp;

void dalib_setup_sectionk (x1, y1, x2, ..., xk, yk)      k = 1,...,4
int x1, y1, x2, ..., xk, yk

void dalib_send_sectionk (to, a)                        k = 1,...,4
int to; unsigned char *a;

void dalib_recv_sectionk (from, a)                      k = 1,...,4
int from; unsigned char *a;

void dalib_copy_sectionk (to, a)                        k = 1,...,4
unsigned char *to; unsigned char *a;
```

7.3 Module 'hostnode'

This module has functions to realize the transfer of sections of distributed arrays to a continguous host aray section and vice versa.

```
     /*********************************************************
     *                                                        *
     *  HOSTARRAY = NODEARRAY (x1:y1, ..., xk:yk)             *
     *                                                        *
     *********************************************************/

void dalib_host_node(k) (a_dsp, a, x1, y1,  ..., xk, yk)
```

```
int *a_dsp;
unsigned char *a;
int *x1, *y1, ..., *xk, *yk;

     /******************************************************
     *                                                     *
     *  NODEARRAY (x1:y1, ....., xk:yk) = HOSTARRAY        *
     *                                                     *
     ******************************************************/

void dalib_node_host(k) (a_dsp, a, x1, y1, .., .., xk, yk)
int *a_dsp;
unsigned char *a;
int *x1, *y1, ..., *xk, *yk;
```

7.4 Module 'replicate'

This module allows the replication of a section of a distributed arrays to a contiguous section of a replicated array.

```
void dalib_replicate(k) (a_dsp, a, ra, x1, y1, ..., xk, yk)
int *a_dsp;
unsigned char *a, *ra;
int *x1, *y1, ..., *xk, *yk;
```

7.5 Module 'cshift'

The module 'cshift' allows circular shifting of distributed arrays. This is only possible for whole arrays.

```
void dalib_cshift (dest_dsp, dest, source, dim, pos)
int *dest_dsp;
unsigned char *dest, *source;
int *dim, *pos;
```

The circular shifting of distributed arrays is only possible for the whole array.

7.6 Movement of Distributed Array Sections

7.6.1 Definition of a Movement

Let A and B be two distributed variables.

$$d : I_A \to P \qquad d : I_B \to P$$

With a data movement some values of B are moved to the variable A. This data movement can be described by a function m.

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

For such a data movement the following data transfers are necessary:

Process p needs from process q: $m^{-1}(d_A^{-1}(p)) \cap d_B^{-1}(q)$

So for every pair (p,q) with $p \neq q$ the following communication statements are necessary:

$$\begin{array}{rll} p : \text{send} & m^{-1}(d_A^{-1}(q)) \cap d_B^{-1}(p) & \text{to q} \\ q : \text{receive} & m^{-1}(d_A^{-1}(q)) \cap d_B^{-1}(p) & \text{from p} \end{array}$$

Example: Communication for Data Movements

```
REAL A(1:20), B(5:17)
...
A(6:13) = B(7:14)
```

A and B are distributed over three processes,

p1 owns `A(1:6)`, `B(5:8)`	needs `B(7)`
p2 owns `A(7:13)`, `B(9:12)`	needs `B(8:14)`
p3 owns `A(14:20)`, `B(13:17)`	needs nothing

The following send and receive statements are generated:

```
p1: send (B(8)) to p2
    A(6) = B(7)
p2: A(8:11)=B(9:12)
    receive (A(7)) from p1
    receive (A(12:13)) from p3
p3: send (B(13:14)) to p2
```

7.6.2 Module 'movement'

For realizing such data movements the routines of this module can be used. This movements have three phases. At first a mapping has to be defined, then the values of the source array have to be send to other processors before values of other processes will be received.

```
/*  Definition of a movement  */

void dalib_move_define (t_dsp, t_low, t_up, s_dsp, s_low, s_up)
int *t_dsp, *t_low, *t_up;
int *s_dsp, *s_low, *s_up;

{ /* maps [s_low:s_up] of source to [t_low:t_up] of target */
```

After defining of a movement values will be send (first subroutine) and then values will be received (second subroutine).

```
void dalib_move_source(k) (a_dsp, a, x1, y1, ..., xk, yk)
int *a_dsp;
unsigned char *a;
int *x1, *y1, ..., *xk, *yk;

void dalib_move_target(k) (a_dsp, a, x1, y1, ..., xk, yk)
int *a_dsp;
unsigned char *a;
int *x1, *y1, ..., *xk, *yk;
```

The following example shows how these functions are used to realize the movement of distributed array sections.

```
c     original program
      real A1 (20)
      real A2 (20,30)
      real A3 (20,10)
      A1 = 10
      A2 = 20
      A1 (2:20) = A1(1:19)
      A3 (5:10,2:9) = A2(3:8,11:18)


c     generated node program

      ...
      call dalib_define_array1 (A1_DSP,4,1,20)
      call dalib_array_pardim (A1_DSP,A1_LOW,A1_HIGH)
      ALLOCATE (A1(A1_LOW:A1_HIGH))
      call dalib_define_array2 (A2_DSP,4,1,20,1,30)
      call dalib_array_pardim (A2_DSP,A2_LOW,A2_HIGH)
      ALLOCATE (A2(1:20,A2_LOW:A2_HIGH))
      call dalib_define_array2 (A3_DSP,4,1,20,1,10)
      call dalib_array_pardim (A3_DSP,A3_LOW,A3_HIGH)
      ALLOCATE (A3(1:20,A3_LOW:A3_HIGH))
      A1 = 10
      A2 = 20
      call dalib_move_define (A1_DSP,2,20,A1_DSP,1,19)
      call dalib_move_source1 (A1_DSP,A1,1,19)
      call dalib_move_target1 (A1_DSP,A1,2,20)
      call dalib_move_define (A3_DSP,2,9,A2_DSP,11,18)
      call dalib_move_source2 (A2_DSP,A2,3,8,11,18)
      call dalib_move_target2 (A3_DSP,A3,5,10,2,9)
      call dalib_undefine_array (A3_DSP)
```

```
DEALLOCATE (A3)
call dalib_undefine_array (A2_DSP)
DEALLOCATE (A2)
call dalib_undefine_array (A1_DSP)
DEALLOCATE (A1)
```

8 Indirect Addressing of Distributed Arrays

The module 'schedule' realizes 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.

8.1 Global Get

```
real A(N), B(M), P(N)

P = ...    ! computed index vector of size N

A = B [P]  ! means:   A[j] = B[P[j]]    for all j  from 1 to N
```

This function can be realized with the following operation:

```
void dalib_global_get1 (a_dsp, b_dsp, p_dsp, a, b, p)
int *a_dsp, *b_dsp, *p_dsp;
int *p;
unsigned char *a, *b;
```

It is also possible to indirect address a two dimension array B.

```
A = B [P,Q]  : A[j] = B[P[j],Q[j]]    for all j

void dalib_global_get2 (a_dsp, b_dsp, p_dsp, q_dsp, a, b, p, q)
int *a_dsp, *b_dsp, *p_dsp, *q_dsp;
int *p, *q;
unsigned char *a, *b;
```

As in many cases not for every element j the indirect access has to be done, there is the possibility to use a mask.

```
void dalib_global_getm1_ (a_dsp, b_dsp, p_dsp, mask_dsp,
                           a,     b,     p,     mask   )
void dalib_global_getm2_ (a_dsp, b_dsp, p_dsp, q_dsp, mask_dsp,
```

```
                        a,    b,    p,    q,    mask     )

int *a_dsp, *b_dsp, *p_dsp, *q_dsp, *mask_dsp;
int *p, *q, *mask;
unsigned char *a, *b;
```

8.2 Global Send

```
   real A(N), B(M), P(N)

   P = ...    ! computed index vector of size N

   B [P]  = A ! means:  B[P[j]] = A[j]   for all j  from 1 to N
```

This function can be realized with the following operation:

```
void dalib_global_set1_ (a_dsp, p_dsp, b_dsp, a, p, b)
int *a_dsp, *b_dsp, *p_dsp;
int *p;
unsigned char *a, *b;
```

The same is possible for a two dimensional array:

```
B [P,Q] = A         !    B[P[j],Q[j]] = A[j]   for all j

void dalib_global_set2 (b_dsp, p_dsp, q_dsp, a_dsp, b, p, q, a)
int *a_dsp, *b_dsp, *p_dsp, *q_dsp;
int *p, *q;
unsigned char *a, *b;
```

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.

```
void dalib_global_setm1_ (op, b_dsp, p_dsp, a_dsp, mask_dsp,
                              b,     p,     a,     mask)

void dalib_global_setm2_ (op, b_dsp, p_dsp, q_dsp, a_dsp, mask_dsp,
                              b,     p,     q,     a,     mask)

int *a_dsp, *b_dsp, *p_dsp, *q_dsp, *mask_dsp;
int *p, *q, *mask, *op;
unsigned char *a, *b;
```

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

8.3 assertions

When using global get and send the following rules haves to be observed:

- all of the arrays must be distributed
- p, q must be an integer*4 array
- a and b must have the same type, no type conversion
- a and p must have same shape and distribution
- Rank of b must be 1 or 2

References