ADCL: Abstract Data and Communication Library
User level API functions

January 22, 2014

Contents

1 Introduction and Terminology

ADCL is an application level communication library aiming at providing the highest possible performance for application level communication patterns, The library provides for each communication pattern a large number of implementations and incorporates a runtime selection logic in order to choose the implementation leading to the highest performance of the application on the current platform. The library provides multiple different runtime selection algorithms, including a brute force search strategy which tests all available implementations of a given communication pattern; a heuristic relying on orthogonal tuning of attributes characterizing an implementation; and a 2k factorial design search strategy designed for large seach spaces and correlated attributes values.

ADCL is not supposed to be a replacement for MPI, but on add-on library. In fact, ADCL relies and exploits many of the features of MPI, and does not offer for example point-to-point transfer primitives. The ADCL API offers high level interfaces of application level collective operations. The high level interfaces are required in order to be able to switch within the library the implementation of the according collective operation without modifying the application itself. Thus, ADCL complements functionality available within MPI. The main objects within the ADCL API are:

- `ADCL_Attribute`s, which are an abstraction for particular characteristic of a function/implementation. Each attribute is represented by the set of possible values for this characteristic.
- `ADCL_Attrset`s, which represent a collection of ADCL attributes
- `ADCL_Function`s, each of them being equivalent to an actual implementation of a particular communication pattern. An ADCL Function can have an attribute-set attached to it, in which case the values for each of the attributes in the attribute-set for this particular function have to be defined.
- `ADCL_Fnctset`s, which represent a collection of ADCL functions providing the same functionality. All Functions in a function-set have to have the same attribute-set. ADCL provides pre-defined function sets, such as for neighborhood communication (`ADCL_FNCTSET_NEIGHBORHOOD`) or for shift operations (`ADCL_FNCTSET_SHIFT`). The user can however also register its own functions in order to utilize the ADCL runtime selection logic.

- `ADCL_Topology` objects, which provides a description of the process topology and neighborhood relations within the application.

- `ADCL_Vector` which specify the data structures to be used during the communication and the actual data. The user can for example *register* a data structure such as a vector or a matrix with the ADCL library, detailing how many dimensions the object has, the extent of each dimension, which parts of the matrix shall be used for communication, the basic datatype of the object, and the pointer to the data array of the object.

- `ADCL_Vmap`: which defines portion of a vector to be used in a particular communication operation/

- `ADCL_Request` objects, which combines a process topology, a functionset and a vector object. The application can initiate a communication by starting a particular ADCL request.

- `ADCL_Timer` object which allows to register a request with a timer object in order to base the tuning of a functionset on a larger code section instead of the function execution time itself. In addition, the timer object allows to co-tune multiple functionsets within the same code sequence.

This document discusses each of the objects in details and explains the user level API functions. Starting from the Fall 2013 release of ADCL, a higher-level API has also been released with provides MPI like interfaces for many of the predefined operations supported by ADCL. In addition, ADCL supports the notion of historic learning for some predefined function sets, i.e. learning across multiple executions of the same operation.

2 Environmental control functions

This section discusses the general functions required to establish the ADCL environment and to shut it down. All ADCL functions return error codes. ADCL leaves it up to the application to take the appropriate actions in case an error occurs. The only exception to that rule is if an error occurs within an MPI function called by ADCL, since MPI's default error behavior is to abort in case of an error. However, the user can change the default behavior of the MPI library by setting the default error handler of `MPI_COMM_WORLD` to `MPI_ERRORS_RETURN` (see also section 7.2 in the MPI-1 [?] specification).

ADCL provides C and F90 interfaces for most functions. The Fortran interface of a routines contains an additional argument compared to its C counterpart, namely the error code. Furthermore, all Fortran ADCL objects are defined as integers, leaning on the approach chosen by MPI. A C application has to include the ADCL header file called `ADCL.h`, a Fortran application has to include the file `ADCL.inc` in any routine utilizing ADCL functions.

2.1 Initializing ADCL

```
int ADCL_Init ( void );

subroutine ADCL_Init ( ierror )
integer ierror
```

`ADCL_Init` initializes the ADCL execution environment. The function allocates internal data structures required for ADCL, and has to be called therefore before any other ADCL function. Upon success, ADCL returns `ADCL_SUCCESS`. It is recommended to call `ADCL_Init` right after `MPI_Init`. It is erroneous to call `ADCL_Init` multiple times.

2.2 Shutting down ADCL

```
int ADCL_Finalize ( void );

subroutine ADCL_Finalize ( ierror )
integer ierror
```

ADCL_Finalize finalizes the ADCL environment. Since the function deallocates internal data structures, it should be called at the very end of the application, but before MPI_Finalize. It is erroneous to call ADCL_Finalize multiple times.

2.3 ADCL program skeletons

Using the two functions described above, the following presents the skeleton for any ADCL application.

```
#include <stdio.h>
#include "mpi.h"
#include "ADCL.h"

int main ( int argc, char **argv )
{
   MPI_Init ( &argc, &argv );
   ADCL_Init ();

   ...
   ADCL_Finalize ();
   MPI_Finalize ();
   return 0;
}
```

Accordingly, the fortran skeleton looks as follows:

```
program ADCLskeleton
   include 'mpif.h'
   include 'adcl.inc'

   integer ierror

   call MPI_Init ( ierror )
   call ADCL_Init (ierror )

   ...
   call ADCL_Finalize ( ierror )
   call MPI_Finalize ( ierror )
end program ADCLskeleton
```

2.4 ADCL error codes

The following is a list of error codes as defined by ADCL.

- `ADCL_SUCCESS` : no error
- `ADCL_NO_MEMORY`: internal memory allocation failed
- `ADCL_ERROR_INTERNAL` : internal ADCL error
- `ADCL_USER_ERROR`: generic user error
- `ADCL_UNDEFINED`: undefined behavior
- `ADCL_NOT_FOUND` : object not found
- `ADCL_INVALID_ARG` : invalid argument passed by user to an ADCL function. Generic error code, only used if one of the codes below do not match.
- `ADCL_INVALID_NDIMS` : invalid number of dimension passed by user to an ADCL function.
- `ADCL_INVALID_DIMS` : invalid dimension passed by user to an ADCL function.
- `ADCL_INVALID_HWIDTH` : invalid number of halo-cells passed by user to an ADCL function.
- `ADCL_INVALID_DAT`: invalid MPI datatype passed by user to an ADCL function.
- `ADCL_INVALID_DATA`: invalid buffer pointer passed by user to an ADCL function.
- `ADCL_INVALID_COMTYPE`: invalid communication type passed by user to an ADCL function.
- `ADCL_INVALID_COMM`: invalid MPI communicator passed by user to an ADCL function.
- `ADCL_INVALID_REQUEST`: invalid ADCL request passed by user to an ADCL function.
- `ADCL_INVALID_NC` : invalid NC argument passed by user to an ADCL function.

- ADCL_INVALID_TYPE: ?
- ADCL_INVALID_TOPOLOGY: invalid ADCL topology passed by user to an ADCL function.
- ADCL_INVALID_ATTRIBUTE: invalid ADCL attribute passed by user to an ADCL function.
- ADCL_INVALID_ATTRSET: invalid ADCL attribute-set passed by user to an ADCL function.
- ADCL_INVALID_FUNCTION: invalid ADCL function passed by user to an ADCL function.
- ADCL_INVALID_WORK_FUNCTION_PTR: invalid ADCL function pointer passed by user to an ADCL function.
- ADCL_INVALID_FNCTSET: invalid ADCL function-set passed by user to an ADCL function.
- ADCL_INVALID_VECTOR: invalid ADCL vector passed by user to an ADCL function.
- ADCL_INVALID_VECTORSET: invalid ADCL vector set passed by user to an ADCL function.
- ADCL_INVALID_DIRECTION: invalid direction argument passed by user to an ADCL function.

3 High level API

The high level API in ADCL imitates the classic MPI collective communications API. It allows the user to bypass the creation of `ADCL_Topology`, `ADCL_Vmap` and `ADCL_Vector` when using the ADCL predefined function-sets. So far, it has been implemented for `ADCL_FNCTSET_IBCAST`, `ADCL_FNCTSET_IALLTOALL`, `ADCL_FNCTSET_ALLTOALL`, `ADCL_FNCTSET_ALLTOALLV`, `ADCL_FNCTSET_REDUCE`, `ADCL_FNCTSET_ALLREDUCE` and `ADCL_FNCTSET_ALLGATHERV` function-sets. When the user calls a high level function, all these steps are handled internally, and a persistent ADCL request is created in the same way as `MPI_Send_init`. Cartesian neighorhood communication is as of today the only operation that still has to use the original and slightly more complex API outlined later in the document.

Once a persistent comminication handle has been created, an actual communication step can be started using either `ADCL_Request_start) for blocking operations or using a ADCL_Request_init) and ADCL_Request_wait for non-blocking operations. For more details on the operations permitted on the request object, please refer to section 9. A request object can also be used in combination with the timer object as explained in section 4.`

3.1 Ibcast

`(Requires a functional implementation of the LibNBC library)`

```
int ADCL_Ibcast_init ( void *buffer, int count,
         MPI_Datatype datatype, int root, MPI_Comm comm,
         ADCL_Request* req);
```

`with`

- `buffer(IN/OUT): starting address of buffer.`
- `count(IN): number of entries in buffer.`
- `datatype(IN): data type of buffer.`
- `root(IN): rank of broadcast root.`
- `comm(IN): MPI communicator.`
- `req(OUT): handle to the newly created ADCL request object.`

3.2 Ialltoall

(Requires a functional implementation of the LibNBC library)

```
int ADCL_Ialltoall_init ( void *sendbuf, int sendcount,
         MPI_Datatype sendtype, void *recvbuf, int recvcount,
         MPI_Datatype recvtype, MPI_Comm comm, ADCL_Request* req)
```

with

- sendbuf(IN): starting address of send buffer.
- sendcount(IN): number of elements to send to each process.
- sendtype(IN): data type of elements of send buffer.
- recvbuf(OUT): address of receive buffer.
- recvcount(IN): number of elements received from any process.
- recvtype(IN): data type of elements of receive buffer.
- comm(IN): MPI communicator.
- req(OUT): handle to the newly created ADCL request object.

3.3 Reduce

```
int ADCL_Reduce_init ( void *sendbuf, void *recvbuf, int count,
         MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
         ADCL_Request* req)
```

with

- sendbuf(IN): starting address of send buffer.
- recvbuf(OUT): address of receive buffer.
- count(IN): number of elements in send buffer.
- datatype(IN): data type of elements.
- op(IN): MPI operation to perform (eg. MPI_SUM).
- root(IN): rank of root process.
- comm(IN): MPI communicator.
- req(OUT): handle to the newly created ADCL request object.

3.4 Allreduce

```
int ADCL_Allreduce_init ( void *sendbuf, void *recvbuf, int count,
         MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
         ADCL_Request* req)
```

with

- sendbuf(IN): starting address of send buffer.
- recvbuf(OUT): address of receive buffer.
- count(IN): number of elements send buffer.
- datatype(IN): data type of elements.
- op(IN): MPI operation to perform (eg. MPI_SUM).
- comm(IN): MPI communicator.
- req(OUT): handle to the newly created ADCL request object.

3.5 Alltoall

```
int ADCL_Alltoall_init ( void *sendbuf, int sendcount,
         MPI_Datatype sendtype, void *recvbuf, int recvcount,
         MPI_Datatype recvtype, MPI_Comm comm, ADCL_Request* req)
```

with

- sendbuf(IN): starting address of send buffer.
- sendcount(IN): number of elements to send to each process.
- sendtype(IN): data type of elements of send buffer.
- recvbuf(OUT): address of receive buffer.
- recvcount(IN): number of elements received from any process.
- recvtype(IN): data type of receive buffer elements.
- comm(IN): MPI communicator.
- req(OUT): handle to the newly created ADCL request object.

3.6 Alltoallv

```
int ADCL_Alltoallv_init ( void *sendbuf, int* sendcnts, int *sdispls,
        MPI_Datatype sendtype, void *recvbuf, int* recvcnts,
        int *rdispls, MPI_Datatype recvtype, MPI_Comm comm,
        ADCL_Request* req)
```

with

- sendbuf(IN): starting address of send buffer.
- sendcnts(IN): integer array equal to the group size specifying the number of elements to send to each process.
- sdispls(IN): integer array (of length group size). Entry j specifies the displacement (relative to sendbuf from which to take the outgoing data destined for process j.
- sendtype(IN): data type of elements of send buffer.
- recvbuf(OUT): address of receive buffer.
- recvcnts(IN): integer array equal to the group size specifying the maximum number of elements that can be received from each process.
- rdispls(IN): integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
- recvtype(IN): data type of receive buffer elements.
- comm(IN): MPI communicator.
- req(OUT): handle to the newly created ADCL request object.

3.7 Allgatherv

```
int ADCL_Allgatherv_init ( void *sendbuf, int sendcount,
        MPI_Datatype sendtype, void *recvbuf, int* recvcnts,
        int *displs, MPI_Datatype recvtype, MPI_Comm comm,
        ADCL_Request* req)
```

with

- sendbuf(IN): starting address of send buffer.
- sendcount(IN): number of elements in send buffer.
- sendtype(IN): data type of elements of send buffer.
- recvbuf(OUT): address of receive buffer.
- recvcnts(IN): integer array (of length group size) containing the number of elements that are to be received from each process.
- rdispls(IN): integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i.
- recvtype(IN): data type of receive buffer elements.
- comm(IN): MPI communicator.
- req(OUT): handle to the newly created ADCL request object.

3.8 Example

The following example shows the usage of the ADCL high level API to broadcast an integer.

```
#include <stdio.h>
#include "mpi.h"
#include "ADCL.h"
#include "nbc.h"

int main ( int argc, char ** argv )
{

    ADCL_Request request;

    MPI_Init ( &argc, &argv );
    ADCL_Init ();

    int data = 10;

    /*****************************************************************/

```

```
    ADCL_Ibcast_init(&data, 1, MPI_INT, 0, MPI_COMM_WORLD, &request);

    /* start the communication. call the following function an
       arbitrary number of times */
    ADCL_Request_start ( request );


    /* if communication is done, free all handles */
    ADCL_Request_free ( &request );

    ADCL_Finalize ();
    MPI_Finalize ();
    return 0;
}
```

4 Timer object

ADCL tunes a functionset by default based on the time spent in the corresponding function. Users might desire to tune a particular communication operation based on a large codesequence, e.g. an entire iteration of their iterative algorithm or similar, well defined sections in the code. There are three specific instance where it makes sense to tune an operation based on a larger code sequence to potentially includes also computational operations:

1. for very short running operations, where clock resolution and synchronization of processes might be an issue.
2. for non-blocking operations, in which (theoratically) the bulk of the communication operation would happen in the background, and thus timing the actual function call is nearly irrelevant.
3. for code sections which contain multiple operations that shall be tuned simultaniously. Tuning each operation separatly can lead in such a scenario to optimal execution time of each operation individually, but not globally from the application perspective.

The timer object allows to register an ADCL_Request with a timer, and add function calls to start and stop the timer outside of the ADCL_Request_start / _init() functions.

```
int ADCL_Timer_create ( int nreq, ADCL_Request *reqs, ADCL_Timer *timer);
```

with

- nreqs(IN): number of requests to be associated with the timer object.
- reqs(IN): array of dimension nreqs containing the array of ADCL_Requests to be associated with the timer object.
- timer(OUT): handle to ADCL timer object.

```
int ADCL_Timer_free ( ADCL_Timer *timer );
```

with

- timer(INOUT): handle to the timer object allocated with ADCL_Timer_create. Upon successful completion, the handle will be set to ADCL_TIMER_NULL.

Important The timer object should be freed before freeing the requests attached to it, otherwise you can get an unexpected behavior in some cases.

```
int ADCL_Timer_start ( ADCL_Timer timer );
int ADCL_Timer_stop ( ADCL_Timer timer );
```

start and stop the timer. The execution time between the start and the stop function will be stored with each ADCL Request that has been associated with the provided timer object.

The following shows an example on how to associate a request with a timer object and how to utilize it in the code.

```
int main ( int argc, char ** argv )
/**********************************************************************/
{
    /* General variables */
    int rank, size, it;
    double datain[10], dataout[10];
    ADCL_Request req;
    ADCL_Timer timer;

    /* Initiate the MPI environment */
    MPI_Init ( &argc, &argv );
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    MPI_Comm_size ( MPI_COMM_WORLD, &size );

    /* Initiate the ADCL library */
    ADCL_Init ();

    /* Initialize datain */
    for (it=0; it <10; it++ ) {
       datain[it] = ...;
       dataout[it] = 0.0;
    }

    /* Initialize a persistent Allreduce operation */
```

```
    ADCL_Allreduce_init ( datain, dataout, 10, MPI_DOUBLE, MPI_SUM,
                          MPI_COMM_WORLD, &req);

    /* define timer object */
    ADCL_Timer_create ( 1, &req, &timer );

    for (it=0; it<MAXIT; it++){
        ADCL_Timer_start( timer );

        /* perform some computation */
        ...
        /* Start the communication */
        ADCL_Request_start ( req1 );

        /* perform some more computation */
        ....
        ADCL_Timer_stop( timer );
    }

    ADCL_Timer_free   ( &timer );
    ADCL_Request_free ( &req1 );

    ADCL_Finalize ();
    MPI_Finalize ();
    return 0;
}
```

5 Attributes and Attribute-sets

An ADCL_Attribute is an abstraction for a particular characteristic of a function/implementation. Each attribute is represented by a set of possible values. An ADCL_Attrset is a group of ADCL attributes. In the following, we present the constructors and destructors for both ADCL objects.

```
int ADCL_Attribute_create ( int maxnvalues, int *array_of_values,
         char ** array_of_value_names, char *attr_name,
         ADCL_Attribute *attr);

subroutine ADCL_Attribute_create ( maxnvalues, array_of_values,
                attr, ierror )
integer maxnvalues, attr, ierr
integer array_of_values (*)
```

with

- maxnvalues(IN): number of possible values for this attribute
- array_of_values(IN): integer array of size maxnvalues containing the possible values for this attribute. The values in this array have to be monotony increasing. However, they do not have to be contiguous.
- array_of_value_names(IN): character array specifying a name for each attribute value. Not available in the fortran interface.
- attr_name(IN): name of the attribute. Not available in the fortran interface.
- attr(OUT): handle to the attribute object

```
int ADCL_Attribute_free   ( ADCL_Attribute *attr );

subroutine ADCL_Attribute_free ( attr, ierror )
integer attr, ierror
```

with

- attr(INOUT): handle to the ADCL attribute to be freed. The handle has to be a valid attribute. Upon successful completion, the handle will be set to ADCL_ATTRIBUTE_NULL.

```
int ADCL_Attrset_create ( int maxnum,
        ADCL_Attribute *array_of_attrs, ADCL_Attrset *attrset );

subroutine ADCL_Attrset_create ( maxnum, array_of_attrs,
               attrset, ierror )
integer maxnum, attrset, ierror
integer array_of_attrs (*)
```

with

- maxnum(IN): number of attributes to be grouped
- array_of_attrs(IN): array of size maxnum containing the handles to attributes. Each entry of the array has to be a valid ADCL attribute created with ADCL_Attribute_create.
- attrset(OUT): handle to the attribute-set object

```
int ADCL_Attrset_free ( ADCL_Attrset *attrset );

subroutine ADCL_Attrset_free ( attrset, ierror )
integer attrset, ierror
```

with

- attrset(INOUT): handle to the ADCL attribute-set to be freed. The handle has to be a valid attribute-set. Upon successful completion, the handle will be set to ADCL_ATTRSET_NULL.

6 Functions and Function-sets

An ADCL_Function is the equivalent to an actual implementation of a particular communication pattern. An ADCL Function can have an attribute-set attached to it, in which case the values for each of the attributes in the attribute-set for this particular function have to be defined. A user can however also decide not to attach an attribute-set to a function by passing in ADCL_ATTRSET_NULL at the particular argument.

An ADCL_Fnctset is a collection of ADCL functions providing the same functionality. All Functions in a function-set have to have the same attribute-set. ADCL provides pre-defined function sets, see section 6.1 for a list of predefined function-sets. The user can also register its own functions in order to utilize the ADCL runtime selection logic.

```
typedef void ADCL_work_fnct_ptr ( ADCL_Request req );

int ADCL_Function_create ( ADCL_work_fnct_ptr *fnctp,
         ADCL_Attrset attrset, int *array_of_attrvalues,
         char *name, ADCL_Function *fnct);

subroutine ADCL_Function_create ( fnctp, attrset,
                array_of_attrvalues, name, fnct, ierror )
external fnctp
integer attrset, fnct, ierror
integer array_of_attrvalues(*)
char name (*)
```

with

- fnctp(IN): function pointer to the actual implementation. The prototype has to be of type ADCL_work_fnct_ptr.
- attrset(IN): valid ADCL attribute-set handle, or ADCL_ATTRSET_NULL. Passing the NULL attribute-set object in forces ADCL to use the brute-force runtime search algorithm.
- array_of_attrvalues(IN): if an attribute-set has been specified, this array of integers has to provide the values for each attribute in the attribute-set.

- name(IN): name for the function. The length of the character string can not exceed ADCL_MAX_NAMELEN. It is allowed to pass in a NULL pointer instead of a string.
- fnct(OUT): handle to the ADCL function object.

```
int ADCL_Function_create_async ( ADCL_work_fnct_ptr *init_fnct,
         ADCL_work_fnct_ptr *wait_fnct,
         ADCL_Attrset attrset, int *array_of_attrvalues,
         char *name, ADCL_Function *fnct);

subroutine ADCL_Function_create_async ( init_fnct, wait_fnct,
                attrset, array_of_attrvalues, name, fnct, ierror )
external init_fnct, wait_fnct
integer attrset, fnct, ierror
integer array_of_attrvalues(*)
char name (*)
```

with

- init_fnct(IN): function pointer to the actual implementation of the initiation function. The prototype has to be of type ADCL_work_fnct_ptr.
- wait_fnct(IN): function pointer to the actual implementation of the completion function. The prototype has to be of type ADCL_work_fnct_ptr.
- attrset(IN): valid ADCL attribute-set handle, or ADCL_ATTRSET_NULL. Passing the NULL attribute-set object in forces ADCL to use the brute-force runtime search algorithm.
- array_of_attrvalues(IN): if an attribute-set has been specified, this array of integers has to provide the values for each attribute in the attribute-set.
- name(IN): name for the function. The length of the character string can not exceed ADCL_MAX_NAMELEN. It is allowed to pass in a NULL pointer instead of a string.
- fnct(OUT): handle to the ADCL function object.

```
int ADCL_Function_free ( ADCL_Function *fnct );
```

```
subroutine ADCL_Function_free ( fnct, ierror )
integer fnct, ierror
```

with

- fnct(INOUT): valid handle to an ADCL function. Upon return, the handle is set to ADCL_FUNCTION_NULL

```
int ADCL_Fnctset_create ( int maxnum, ADCL_Function *fncts,
         char *name, ADCL_Fnctset *fnctset );
```

```
subroutine ADCL_Fnctset_create ( maxnum, fncts, name,
                fnctset, ierror )
integer maxnum, fnctset, ierror
integer fncts(*)
char name (*)
```

with

- maxnum(IN): number of ADCL functions to be bundled to a function set
- fncts(IN): array of size maxnum containing the handles to the ADCL functions. All functions have to provide the same attribute-set.
- name(IN): name of the function set. It is allowed to pass in a NULL pointer instead of a name. The length of the character string can not exceed ADCL_MAX_NAMELEN.
- fnctset(OUT): handle for the ADCL function-set.

The following function provides a short cut for scenarios, where a single function shall be executed with different attribute values. Thus, instead of the user having to create all the individual funtions and register them with ADCL, this interface provides the opportunity to specify the function pointer, an attribute set, and get an Functionset handle back. ADCL will create internally a function with the same function pointer for each possible combination of attribute values. Some combinations of attribute values can be also excluded.

```
int ADCL_Fnctset_create_single ( ADCL_work_fnct_ptr *init_fnct,
         ADCL_work_fnct_ptr *wait_fnct, ADCL_Attrset attrset,
         char *name, int **without_attr_combinations,
         int num_without_attr_combinations,
         ADCL_Fnctset *fnctset );

subroutine ADCL_Fnctset_create_single ( void *init_fnct,
                void *wait_fnct, int *attrset, char *name,
                int *without_attr_vals,
                int *num_without_attr_vals,
                int *fnctset, int *ierror, int name_len );
external init_fnct, wait_fnct
integer attrset, without_attr_vals, num_without_attr_vals
integer fnctset, ierror
char name (*)
```

with

- init_fnct(IN): function pointer to the actual implementation of the init function. The prototype has to be of type ADCL_work_fnct_ptr.
- wait_fnct(IN): function pointer to the actual implementation of the completion function. The prototype has to be of type ADCL_work_fnct_ptr. It is allowed to pass in a NULL pointer if there is no need to a completion function.
- attrset(IN): valid ADCL attribute-set handle, or ADCL_ATTRSET_NULL. It is illegal for this function to pass in the NULL attribute-set handle.
- name(IN): name for the function. The length of the character string can not exceed ADCL_MAX_NAMELEN. It is allowed to pass in a NULL pointer instead of a string.
- without_attr_vals(IN): array containing the attribute values that the user want to exclude.
- num_without_attr_vals(IN): number of entries in the array containing the attribute values that the user want to exclude.
- fnctset(OUT): handle to the ADCL function-set object.

```
int ADCL_Fnctset_free   ( ADCL_Fnctset *fnctset );

subroutine ADCL_Fnctset_free ( fnctset, ierror )
integer fnctset, ierror
```

with

- fnctset(INOUT): valid ADCL function-set handle to be freed.
  After successful completion, the handle is set to ADCL_FNCTSET_NULL.

6.1 Predefined Function-sets

ADCL currently supports the following predefined function-sets:

- ADCL_FNCTSET_NEIGHBORHOOD: a function-set supporting n-dimensional neighborhood communication. The dimension of the neighborhood communication is determined by the dimension of the vector object used during the request creation time.
- ADCL_FNCTSET_ALLGATHERV: a function-set supporting an MPI_Allgatherv style communication.
- ADCL_FNCTSET_ALLREDUCE: a function-set supporting an MPI_Allreduce style communication.
- ADCL_FNCTSET_REDUCE: a function-set supporting an MPI_Reduce style communication.
- ADCL_FNCTSET_ALLTOALL: a function-set supporting an MPI_Alltoall style communication.
- ADCL_FNCTSET_ALLTOALLVV: a function-set supporting an MPI_Alltoallv style communication.
- ADCL_FNCTSET_IBCAST: a function-set supporting a non-blocking broadcast operation similarly to MPI_Ibcast. Requires a functional implementation of the LibNBC library.
- ADCL_FNCTSET_IALLTOALL: a function-set supporting a non-blocking all-to-all operation similarly to MPI_IAlltoall. Requires a functional implementation of the LibNBC library.

Other predefined function-sets might be added in following releases of ADCL.

6.2 Examples

The following is an example for a code registering three functions without an attribute-set, and combining them to a function-set. Since there is no attribute-set attached to the functions, the array of attribute-values argument of the ADCL function constructors can be a NULL pointer. Please note, that not setting an attribute-set forces ADCL to fall back to the brute force runtime selection logic.

```
#include <stdio.h>
#include "ADCL.h"
#include "mpi.h"

void test_func_1 ( ADCL_Request req );
void test_func_2 ( ADCL_Request req );
void test_func_3 ( ADCL_Request req );

int main ( int argc, char ** argv )
{
    ADCL_Function funcs[3];
    ADCL_Fnctset fnctset;
    int i;

    MPI_Init ( &argc, &argv );
    ADCL_Init ();

    ADCL_Function_create ( (ADCL_work_fnct_ptr *)test_func_1,
         ADCL_ATTRSET_NULL, NULL, "test_func_1", &(funcs[0]));
    ADCL_Function_create ( (ADCL_work_fnct_ptr *)test_func_2,
         ADCL_ATTRSET_NULL, NULL, "test_func_2", &(funcs[1]));
    ADCL_Function_create ( (ADCL_work_fnct_ptr *)test_func_3,
         ADCL_ATTRSET_NULL, NULL, "test_func_3", &(funcs[2]));

    ADCL_Fnctset_create ( 3, funcs, "trivial functions",
         &fnctset );

    /* Do something with the fnctset */

    ADCL_Fnctset_free ( &fnctset );
    for ( i=0; i<3; i++ ) {
       ADCL_Function_free ( &funcs[i] );
    }
    ADCL_Finalize ();
    MPI_Finalize ();
    return 0;
}

void test_func_1 ( ADCL_Request req ) {
```

```
    int rank;

    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    printf("%d: In test_func_1 \n", rank);
    return;
}

void test_func_2 ( ADCL_Request req ) {
    int rank;

    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    printf("%d: In test_func_2\n", rank);
    return;
}

void test_func_3 ( ADCL_Request req ) {
    int rank;

    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    printf("%d: In test_func_3\n", rank);
    return;
}
```

7 Topology

An ADCL_Topology objects contains the description of the process topology and neighborhood relations within the application. The reason ADCL introduces this abstraction and does not rely entirely on the MPI cartesian communicators is, that many codes have a multi-dimensional process distribution, where the process dimensions are running in a different order compared to a cartesion MPI communicator. Thus, ADCL provides a topology constructor using MPI cartesion communicators, but also a generic constructor, where the user can define for each process who its neighbors are.

```
int ADCL_Topology_create  ( MPI_Comm cart_comm,
    ADCL_Topology *topo);

subroutine ADCL_Topology_create ( cart_comm, topo, ierror )
integer cart_comm, topo, ierror
```

with

- cart_comm(IN): MPI cartesian communicator. A call to MPI_Topo_test has to return MPI_CART for cart_comm to be valid in this function call.
- topo(OUT): handle of an ADCL topology object.

```
int ADCL_Topology_create_generic ( int ndims, int *lneighbors,
    int *rneighbors, int *coords, int direction,
    MPI_Comm comm, ADCL_Topology *topo);

subroutine ADCL_Topology_create_generic ( ndims, lneighbors,
    rneighbors, coords, direction, comm, topo, ierror )
integer ndims, direction, comm, topo, ierror
integer lneighbors(*), rneighbors(*), coords(*)
```

with

- ndims(IN): number of dimensions of the process topology.
- lneighbors(IN): integer array of dimension ndims containing the ranks of the left neighbors for each dimension. In case

a left neighbor does not exist (e.g. the process is at the boundary of the process topology), the according entry in the array has to be set to MPI_PROC_NULL.

- rneighbors(IN): integer array of dimension ndims containing the ranks of the right neighbors for each dimension. In case a right neighbor does not exist (e.g. the process is at the boundary of the process topology), the according entry in the array has to be set to MPI_PROC_NULL.
- coords(IN): integer array of dimension ndims containing the coordinates of the process in the process topology.
- direction(IN): ?
- comm(IN): valid MPI communicator. In contrary to the previous function, this does not have to be an MPI cartesian communicator.
- topo(OUT): handle of an ADCL topology object.

```
int ADCL_Topology_free ( ADCL_Topology *topo );
```

with

- topo(INOUT): ADCL topology object to be released. Upon successful completion, the handle will be set to ADCL_TOPOLOGY_NULL.

7.1 Example

The following example shows how to register a 2-D process topology with ADCL for an arbitrary number of processes using MPI cartesian communicators.

```
#include <stdio.h>
#include "ADCL.h"
#include "mpi.h"

int main ( int argc, char ** argv )
{
    MPI_Comm cart_comm;
    ADCL_Topology topo;
```

```
    int cdims[]={0,0}, periods[]={0,0};

    MPI_Init ( &argc, &argv );
    ADCL_Init ();

    MPI_Dims_create ( size, 2, cdims );
    MPI_Cart_create ( MPI_COMM_WORLD, 2, cdims, periods, 0,
                      &cart_comm);
    ADCL_Topology_create ( cart_comm, &topo );

    /* do something useful with the topology object */

    ADCL_Topology_free ( &topo );
    MPI_Comm_free ( &cart_comm );

    ADCL_Finalize ();
    MPI_Finalize ();
    return 0;
}
```

8 Vectors and Vector-maps

An ADCL_Vector combined with the ADCL_Vmap specifies the data structures to be used during the communication and the actual data.

For the collectives defined in the MPI standard it is noticed that the arguments of the parameters can be separated into three groups: information concerning the data (buffer, data type), concerning the process topology (communicator, root) and related to the communication pattern (element counts, reduction operation, array of element counts or displacements). As an example, for the MPI_Bcast interface, the information about the data consists of the buffer and the data type, root and the MPI communicator comm give information about the process topology and count is related to the communication pattern. As each of the parameters of the MPI collectives falls into one of these three groups, they form the basis of the ADCL objects: information concerning the data is stored in the ADCL vector object, information concerning the process topology in the ADCL topology object and information related to the communication pattern becomes part of the new vmap object. As of today, there are five different Vmap types defined in ADCL:'

- HALO: specifies the number of halo-cells in each direction (hwidth).
- ALL: a value specifiying the number of elements used in the communication (count).
- REDUCE: a value specifying the number of elements used in the communication (count) along with an operation (op) to be performed on the data.
- LIST: an array of values specifying the number of elements communicated to each process, with the element at position i in the array used to communicate to rank i. In addition, an array of displacements has to be provided following the same rules as for the array of counts arguments.
- INPLACE: an abstract vmap object that can be used similarly to MPI_INPLACE as an argument to some function sets.

The following are the (brief) interfaces to allocate each Vmap object. For the sake of brevity we do not detail all arguments at this point.

```
int ADCL_Vmap_halo_allocate ( int hwidth, ADCL_Vmap *vec );
int ADCL_Vmap_list_allocate ( int size, int* rcnts, int* displ,
                              ADCL_Vmap *vec );
int ADCL_Vmap_allreduce_allocate ( MPI_Op op, ADCL_Vmap *vec );
int ADCL_Vmap_reduce_allocate ( MPI_Op op, ADCL_Vmap *vec );
int ADCL_Vmap_alltoall_allocate ( int scnt, int rcnt, ADCL_Vmap *vec );
int ADCL_Vmap_all_allocate ( ADCL_Vmap *vec );
int ADCL_Vmap_inplace_allocate ( ADCL_Vmap *vec );
int ADCL_Vmap_free  ( ADCL_Vmap *vec );
```

Similarly to all other objects, the Vmap object can be freed after utilizion using

```
int ADCL_Vmap_free ( ADCL_Vmap *vmap )
```

ADCL distinguishes between allocating a vector and registering a vector. The difference is, that in the first case, the library allocates the memory for the object as specified by the user, while in the second case the memory has to be allocated by the application.

```
int ADCL_Vector_allocate_generic ( int ndims, int *dims, int nc,
                           ADCL_Vmap vmap, MPI_Datatype dat,
                           void *data, ADCL_Vector *vec )
```

with

- ndims(IN): number of dimension of the data structure excluding the nc argument detailed below.
- dims(IN): array of dimension ndims containing the extent of each dimension including the halo-cells.
- nc(IN): number of elements per entry. Since many scientific codes have matrices where each entry of the matrix is a submatrix itself, this argument gives the possibility to specify the dimension of the submatrix.
- vmap(IN): vector map to be used for this vector.
- dat(IN): basic MPI datatype describing the data type of the matrix

- data(OUT): pointer to the buffer allocated. This pointer will be required for calculations within the user code, since the buffer has been allocated by ADCL. ADCL guarantees, that a contiguous memory location has been allocated for multi-dimensional arrays. Please note, that the buffer pointer is **not** the beginning of the data array, but the pointer to the multi-dimensional matrix itself. As an example if ndims=2 and the temporary variable used within ADCL to allocate the 2-D array is called tmp_matrix, then the buffer pointer returned in this argument is data = tmp_matrix, which is not equal to data = &(tmp_matrix[0][0]) in C!

- vec(OUT): handle to ADCL vector object.

There is no Fortran interface defined for this routine. For a discussion, why no Fortran interface is provided for this routine, please refer to the discussion of MPI_Alloc_mem in the MPI-2 [?] specification section 4.1.1.

```
int ADCL_Vector_register_generic ( int ndims, int *dims, int nc,
                                   ADCL_Vmap vmap, MPI_Datatype dat,
                                   void *data, ADCL_Vector *vec )

subroutine ADCL_Vector_register_generic ( ndims,  dims, nc,
                vmap, dat, data, vec, ierror )
integer ndims, nc, vmap, dat, vec, ierror
integer dims(*)
TYPE data(*)
```

with

- ndims(IN): number of dimension of the data structure excluding the nc argument detailed below.

- dims(IN): array of dimension ndims containing the extent of each dimension including the halo-cells.

- nc(IN): number of elements per entry. Since many scientific codes have matrices where each entry of the matrix is a submatrix itself, this argument gives the possibility to specify the dimension of the submatrix.

- vmap(IN): variable describing which parts of the matrix can be used for communication. The Vmap object is typically the result of an ADCL_Vmap_allocate* function as shown above.
- dat(IN): basic MPI datatype describing the data type of the matrix
- data(IN): pointer to the data array. Please note, that the buffer pointer has to be the pointer to the multi-dimensional matrix itself, and **not** the beginning of the data array. As an example if ndims=2 and the variable used within application is defined as double tmp_matrix[10][10], the buffer pointer passed to ADCL has to be tmp_matrix, which is not equal to &(tmp_matrix[0][0]) in C!
- vec(OUT): handle to ADCL vector object.

```
int ADCL_Vector_free ( ADCL_Vector *vec );
```

with

- vec(INOUT): handle to the vector object allocated with ADCL_Vector_allocate. It is illegal to call this function with a vector object registered with ADCL_Vector_register. Upon successful completion, the handle will be set to ADCL_VECTOR_NULL.

There is no Fortran interface defined for this routine. For a discussion, why no Fortran interface is provided for this routine, please refer to the discussion of MPI_Alloc_mem in the MPI-2 [?] specification section 4.1.1.

```
int ADCL_Vector_deregister ( ADCL_Vector *vec );

subroutine ADCL_Vector_deregister ( vec, ierror )
integer vec, ierror
```

with

- vec(INOUT): handle to the vector object registered with ADCL_Vector_register. It is illegal to call this function with a vector object allocated with ADCL_Vector_allocate. Upon successful completion, the handle will be set to ADCL_VECTOR_NULL.

The following example assumes, that each entry of this matrix contains of a vector with five elements.

```
#include <stdio.h>
#include "ADCL.h"
#include "mpi.h"

/* Dimensions of the data matrix per process */
#define DIM0  8
#define DIM1  4

int main ( int argc, char ** argv )
{
    int dims[2];
    int nc=5, hwidth=1;
    double ***data;
    ADCL_Vector vec;
    ADCL_Vmap vmap;

    MPI_Init ( &argc, &argv );
    ADCL_Init ();

    dims[0] = DIM0 + 2*hwidth;
    dims[1] = DIM1 + 2*hwidth;
    ADCL_Vmap_halo_allocate ( hwidth, &vmap);
    ADCL_Vector_allocate_generic ( 2,  dims, nc, vmap,
        MPI_DOUBLE, &data, &vec );

    /* now you can access the elements of the vector in order to perform
       computations, e.g */
    for ( i=1; i<DIM0; i++ ) {
       for ( j=1; j<DIM1; j++) {
         for ( k=0; k < nc; k++ ) {
           data[i][j][k] = ...
       }
    }

    ADCL_Vector_free ( &vec );
    ADCL_Vmap_free ( &vmap );
    ADCL_Finalize ();
```

```
    MPI_Finalize ();
    return 0;
}
```

9 Requests

An ADCL_Request object combines a process topology, a function-set and a vector object to a single user-level handle. Using this handle, the application can initiate a communication by starting a particular ADCL request. In the following we will detail request constructors and destructors, the available functions for initiating a communication, and reflection functions in order to retrieve certain components of an ADCL request from a user level function.

9.1 Constructors and Destructors

```
int ADCL_Request_create ( ADCL_Vector vec, ADCL_Topology topo,
         ADCL_Fnctset fnctset, ADCL_Request *req );

subroutine ADCL_Request_create ( vec, topo, fnctset, req, ierror)
integer vec, topo, fnctset, req, ierror
```

with

- vec(IN): vector object identifying the data arrays used for the communication. The vector object can be ADCL_VECTOR_NULL for a user defined function-set following the requirements and restrictions of the according function-set. However, for the predefined function-sets, the vector object has to be provided following the dimension matching rule explained below.
- topo(IN): topology object identifying the neighboring processes. The topology object can be ADCL_TOPOLOGY_NULL for user defined function-set following the requirements and restrictions of the according function-set. However, for the predefined function-sets, the topology object has to be provided following the dimension matching rule explained below.
- fnctset(IN): ADCL function-set to be evaluated
- req(OUT): handle to the newly created ADCL request object

Dimension matching rule: The dimension of the vector object has to be equal to the dimension of the process topology. The user has one additional degree of freedom by using the nc argument in vector creation, which allows to match a vector of dimension $n+1$ to a n dimensional process topology, by not distributing the last dimension of the vector.

Remark on the dimension matching rule: It is fairly straight forward to define a more generic vector object, where the user could define a multi-dimensional vector and assign to each of the dimensions a flag which indicates whether the data is distributed accordingly in this dimension or not. The solution applied so far restricts this approach by reducing the dimension of a matrix which is not distributed across the processes to the last dimension. This is due to our findings, that it is a common approach in many codes. However, if an end-user faces a problem requiring the generic solution, please contact the authors of ADCL. In any case, the number of dimension of the matrix distributed across the process has to match the number of dimensions of the process topology.

```
int ADCL_Request_create_generic ( ADCL_Vectset vecset,
         ADCL_Topology topo, ADCL_Fnctset fnctset,
         ADCL_Request *req );

subroutine ADCL_Request_create_generic ( vecset, topo,
                fnctset, req, ierror )
integer vecset, topo, fnctset, req, ierror
```

with

- vecset(IN): vector set to be used in the request. The extent of the array of send vectors and receive vectors has to match the number of dimensions of the topology object.
- topo(IN): topology object identifying the neighboring processes. The topology object can be ADCL_TOPOLOGY_NULL for user defined function-set following the requirements and restrictions of the according function-set. However, for the predefined function-sets, the topology object has to be provided following the dimension matching rule explained below.
- fnctset(IN): ADCL function-set to be evaluated
- req(OUT): handle to the newly created ADCL request object

```
int ADCL_Request_free ( ADCL_Request *req );

subroutine ADCL_Request_free ( req, ierror )
integer req, ierror
```

with

- req(INOUT): valid ADCL request handle. After successful completion, the handle will be set to ADCL_REQUEST_NULL.

9.2 Initiating communication operations

The following function can be used to start a blocking instance of the communication pattern.

```
int ADCL_Request_start ( ADCL_Request req );

subroutine ADCL_Request_start ( req, ierror )
integer req, ierror
```

with

- req(IN): valid ADCL request

The following two functions represent a non-blocking version of ADCL_Request_start. There are no restrictions in ADCL on how many ADCL requests can be ongoing simultaneously. However, for a single request a call to ADCL_Request_init has to be followed by a call to ADCL_Request_wait.

```
int ADCL_Request_init  ( ADCL_Request req );

subroutine ADCL_Request_init ( req, ierror )
integer req, ierror
```

with

- req(IN): valid ADCL request

```
int ADCL_Request_wait  ( ADCL_Request req );

subroutine ADCL_Request_wait ( req, ierror )
integer req, ierror
```

with

- req(IN): valid ADCL request started with ADCL_Request_init.

9.3 Retrieving internal components

The following function helps to retrieve the communicator object assigned to a request. It is intended to be used by user registered ADCL functions internally. Similar functions for retrieving data pointers will be introduced soon

```
int ADCL_Request_get_comm  ( ADCL_Request req,
         MPI_Comm *comm, int *rank, int *size );


subroutine ADCL_Request_get_comm ( req, comm, rank,
                size, ierror )
integer req, comm, rank, size, ierror
```

with

- req(IN): valid ADCL request handle
- comm(OUT): communicator used when creating the according ADCL topology object
- rank(OUT): rank of the process in the according communicator comm.
- size(OUT): size of the communicator comm.

```
int ADCL_Request_get_curr_function ( ADCL_Request req,
         char **function_name, char ***attribute_names,
         int *num_of_attributes, char ***attribute_values_names,
         int **attribute_values );
```

with

- req(IN): valid ADCL request handle
- function_name(OUT): Name of the function currently being called. This string has to be freed by the user level function.
- attribute_names(OUT):
- num_of_attributes(OUT): number of attributes. This value is equal to the extent of the attribute_names, attribute_values_names and attribute_values arrays.
- attribute_values_names(OUT): array of names for the attribute values of the current function.
- attribute_values(OUT): array of integers containing the attribute values of the current function.

9.4 Save/Restore request status

The following function helps the user to save the progress in the search process of the fastest implementation when using the brute force strategy. Then, the user can simply resume the search from where he stopped by restoring the request status using the function ADCL_Request_restore_status.

```
int ADCL_Request_save_status ( ADCL_Request req, int *tested_num,
          double **unfiltered_avg, double **filtered_avg,
          double **outliers, int *winner_so_far );
```

with

- req(IN): valid ADCL request handle
- tested_num(OUT): number of already evaluated ADCL functions.
- unfiltered_avg(OUT): unfiltered average of the evaluated ADCL functions.
- filtered_avg(OUT): filtered average of the evaluated ADCL functions.

- outliers(OUT): number of outliers detected in the evaluated ADCL functions.
- winner_so_far(OUT): the winning implementation so far.

```
int ADCL_Request_restore_status ( ADCL_Request req, int tested_num,
         double *unfiltered_avg, double *filtered_avg,
         double *outliers );
```

with

- req(IN): valid ADCL request handle
- tested_num(IN): number of already evaluated ADCL functions.
- unfiltered_avg(IN): unfiltered average of each evaluated ADCL functions.
- filtered_avg(IN): filtered average of each evaluated ADCL functions.
- outliers(IN): number of outliers detected in the evaluated ADCL functions.

9.5 Examples

The following example demonstrates the usage of the predefined function set ADCL_FNCTSET_NEIGHBORHOOD for a three dimensional matrix and process topology.

```
#include <stdio.h>
#include "ADCL.h"
#include "mpi.h"

/* Dimensions of the data matrix per process */
#define DIM0  64
#define DIM1  32
#define DIM2  32

int main ( int argc, char ** argv )
{
    int hwidth, dims[3];
    double ***data;
    int cdims[]={0,0,0};
```

```
    int periods[]={0,0,0};

    MPI_Comm cart_comm;
    ADCL_Vector vec;
    ADCL_Topology topo;
    ADCL_Request request;

    MPI_Init ( &argc, &argv );
    ADCL_Init ();

    MPI_Dims_create ( size, 3, cdims );
    MPI_Cart_create ( MPI_COMM_WORLD, 3, cdims, periods, 0, &cart_comm);
    ADCL_Topology_create ( cart_comm, &topo );

    /*******************************************************************/
    /* Test 1: hwidth=1, nc=0 */
    hwidth=1;
    dims[0] = DIM0 + 2*hwidth;
    dims[1] = DIM1 + 2*hwidth;
    dims[2] = DIM2 + 2*hwidth;
    ADCL_Vector_allocate ( 3,  dims, 0, hwidth, MPI_DOUBLE, &data, &vec );
    ADCL_Request_create ( vec, topo, ADCL_FNCTSET_NEIGHBORHOOD, &request );

    /* start the computation and communication. call the following function
       an arbitrary number of times */
    ADCL_Request_start ( request );


    /* if computation/communication is done, free all handles */
    ADCL_Request_free ( &request );
    ADCL_Vector_free ( &vec );
    ADCL_Topology_free ( &topo );
    MPI_Comm_free ( &cart_comm );

    ADCL_Finalize ();
    MPI_Finalize ();
    return 0;
}
```