Specification of Advanced Communication Primitives (ACP) Library

Version 1.0 (draft)

1 September, 2014

Acknowledgements

This manual has been written as a product of Advanced Computing Environment (ACE) Project. This project is supported by CREST program of Japan Science and Technology Agency (JST).

Organizations and Members of ACE Project:

- Kyushu University
  Takeshi Nanri, Toshiya Takami, Ryutaro Susukita, Hiroaki Honda, Taizo Kobayashi and Yoshiyuki Morie

- Fujitsu Ltd.
  Shinji Sumimoto, Yuichiro Ajima, Kazushige Saga, Naoyuki Shida and Takafumi Nose

- ISIT Kyushu
  Hidetomo Shibamura and Takeshi Soga

- Kyoto University
  Keiichiro Fukazawa

Contents

1. Introduction

1.1 About This Document

This document introduces the specifications of the interfaces of Advanced Communication Primitives (ACP) Library. This library is designed to enable applications with sufficient inherent parallelism to achieve high scalability up to exa-scale computing systems, where the number of processes is expected to be more than a million.

1.2 Motivation

As the performance of environments for high-performance computing (HPC) is approaching from peta to exa, the number of cores in one node is increasing, while the amount of memory per node is expected to remain almost the same. Therefore, reducing memory consumption has become an important issue for achieving sustained scalability toward exa-scale computers.

Especially, communication middleware on those environments must be carefully designed to achieve high memory efficiency. There is always a trade-off between the memory consumption and the performance of communication. To avoid conflicts of resources and achieve high performance, sufficient amount of buffers should be allocated. Therefore, appropriate control of the allocation and de-allocation of buffers is a key to enable memory-efficient communications.

1.3 Approach

To minimize the requirements of memory consumption at the initialization, ACP Library provides RDMA model as the basic communication layer. On interconnect networks where RDMA is supported as a fundamental facility, this layer can be implemented with minimal memory consumption and overhead.

Since programming with RDMA needs detailed operations such as memory registrations, address exchanges and synchronizations, ACP also prepares some sets of programmer-friendly interfaces as the middle layer. To enable the library to consume just-enough amount of memory, each interface of the middle layer requires explicit allocation of the memory region before using it. This region can be explicitly de-allocated so that the memory region can be reused for other purposes.

Each of the interfaces of this middle layer is primitive and independent. For example, the channel interface in this layer only supports one-directional and in-order data transfer between a pair of processes. This helps to minimize the memory consumption and overhead in the implementation. Also, the independent interfaces enable precise control of the allocation and de-allocation of memory regions for them. At this point, interfaces of channels, vectors, lists and memory allocation are defined in ACP. There are plans for other interfaces such as group communications, deques, maps, sets and counters.

1.4 Terminology

process	A unit of program execution by OS.
task	A unit of parallel program execution including multiple processes.
rank	A number to distinguish process in a task.
ACE project	Advanced Communication for Exa project. It develops a communication library with low memory consumption and low overhead.
ACP library	Advanced Communication Primitives library. A library that has been developed in the ACE project.
GMA	Global Memory Access. Direct access to the memory of a remote process in a task.
global memory	Memory region to which remote processes can establish GMAs.
starter memory	A region of global memory that is prepared in each rank at the initialization.
global address	64bit address space shared by all of the processes in a task.
address translation key	An identifier used for translating logical address to global address.
color	A number to distinguish the path for transferring data with GMA.
channel	Virtual path to transfer messages between a pair of processes.
vector	One-dimensional array.
list	Bi-directional list.
iterator	An index to refer to an element in a data structure.

2. Overview

2.1 Fundamental Design

2.1.1 Execution Model

The execution model of ACP is a single program multiple data (SPMD) model. Therefore, each process in a task executes the same program. Processes are distinguished by ranks. Ranks are mapped to the processes at the initialization. ACP also supports re-mapping ranks during the execution.

2.1.2 Communication Models

As written in the previous section, the basic communication model of ACP is RDMA. However, in this model, programmers need to write codes for memory registration, address exchange and appropriate synchronization before establishing data transfer.
Therefore, ACP also supports additional two categories of interfaces as the middle layer, the communication patterns and the data structures. The communication pattern interfaces support explicit transfers of data on the local memory of each process. Currently, ACP provides one interface, channel, as this category of interfaces.
On the other hand, the data structure interface supports implicit data transfer via the operations on dynamic distributed data structures. At this point, vector and list are supported in this category. The heart of this category is an asynchronous global memory allocator that allocates a segment of global memory involving no processor at the provider-side. The design concept of this category derives from the C/C++ languages and is more straightforward than the novel design concept of the basic layer.
As mentioned previously, memory efficiency is the target property in the design of ACP. Towards this target, each interface in the middle layer is designed to support a primitive facility so that it can be implemented with minimal memory consumption and overhead.
In addition to that, to enable just-enough memory consumption, each interface of the middle layer consists of functions for allocation and de-allocation. Programs must explicitly call the allocation function of the interface before establishing data-transfer operations in it. This function allocates a region of memory to be used by a set of processes that participate in the corresponding communication pattern or the data structure. Eventually the information required for establishing communications on the interface is exchanged until the completion of the first data transfer with it. On the other hand, the de-allocation function frees the memory regions allocated for the

interface.
In a program, the same set of processes can call the allocation function of the interface multiple times. This will prepare independent memory region for each call. This policy can cause redundant allocation of memory region. However, sharing resources among multiple calls of allocation can complicates the mechanisms for handling memory regions and causes higher memory consumptions and overheads.
To establish communications that are not supported in the middle layer, the programmers can use the interfaces of the basic RDMA operations of ACP. It consists of two categories of interfaces, the global memory management (GMM) and the global memory access (GMA). In this model, any local memory regions can be mapped to the global address space shared among processes. ACP uses 64bit unsigned integer as the global address that specifies the position in the space. As GMA, the remote copy operation and some atomic operations are supported.

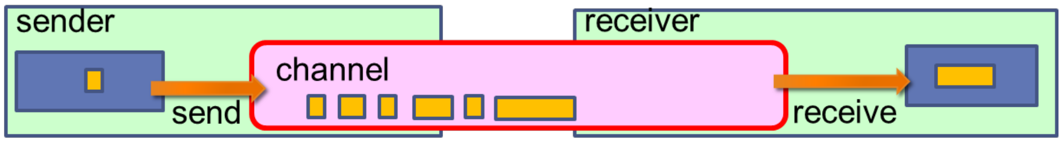
2.2 Interfaces

2.2.1 Infrastructure

Each process starts its execution as a part of an ACP program by calling the initialization function, **`acp_init`**, and ends by calling finalization function, **`acp_finalize`**. ACP also prepares a function for aborting the program, **`acp_abort`**, in case of an error.
The reset function, **`acp_reset`**, remaps ranks to the processes as specified by its argument. There are functions to query current rank of the process, **`acp_rank`**, and the total number of processes, **`acp_procs`**.
Processes are implicitly synchronized at the initialization, reset and finalization. There is also a synchronization function, **`acp_sync`**, to explicitly synchronize all of the processes. This function returns after determining arrivals of all processes.

2.2.2 Channel

This interface provides a virtual pass for message passing from one process to another. The data transfer supported by it is single direction and in-order. Therefore, the implementation of it can be simple and efficient. Before establishing data transfer with the interface, both the sender process and the receiver process must call the allocation function to establish a channel between them. If a channel has become useless, the program can free the channel by calling the de-allocation function at both sides.



The allocation function, **acp_create_ch**, of this interface allocates a region of memory according to the role of the process that invoked the function. After that, it starts exchanging the information about the region with another peer of the channel to establish the connection of the channel. Then, it returns a handle of the channel, with the data type of **acp_ch_t**, without waiting for the completion of the connection. The library implicitly progresses the establishment of the connection.

The non-blocking functions for sending and receiving messages, **acp_nbsend_ch** and **acp_nbrecv_ch**, on a channel can be invoked before the completion of its connection. Each of these functions stores the information about the invocation in the request queue of the channel, and returns a request handle with the data type of **acp_request_t**. It starts transferring messages after it has completed the connection. The wait function, **acp_wait_ch**, wait for the completion of the request.

The non-blocking de-allocation function, **acp_free_ch**, starts freeing memory regions of the channel specified by the handle. The behavior of the functions of send and receive on the channel that has been specified for this function is not defined.

2.2.3 Vector

The vector interface provides the data structure and algorithms of dynamic array. The type of for referencing vectors is **acp_vector_t**, and the type of the iterator of vectors is **acp_vector_it_t**. Vectors can be created via the constractor function, **acp_create_vector**, or the duplication function, **acp_duplicate_vector**. As an argument, each of these functions accepts the rank to place the instance of the vector.

2.2.4 List

The list interface provides the data structure and algorithms of bidirectional linked list. The type for referencing lists is **acp_list_t**, and the type for the iterator of lists is **acp_list_it_t**. Lists are created via the constructor function, **acp_create_list**. This function prepares an empty list. As an argument, it accepts the rank to place the control structure of the list. Elements can be added to a list via functions, such as **acp_push_back_list** and **acp_insert_list**. Each of these functions also accepts the rank to allocate the instance of the element, as an argument. Therefore, the control structure and the elements of a list can be placed on different processes.

2.2.5 Malloc

The malloc interface provides the functions for asynchronous allocation and de-allocation of global memory. It is the same as that of the asynchronous global heap that designs a memory pool to be accessible via RDMA as well as via processor instructions. This interface consists of the **`acp_malloc`** function that allocates a segment of global memory from specified process by a rank number. A global memory segment allocated by this function can be freed by the **`acp_free`** function.

2.2.6 Global Memory Management

The basic layer provides global address space shared among all of the processes. This space is addressed by 64bit unsigned integer, so that it can be directly handles by the atomic operations of CPUs and devices.

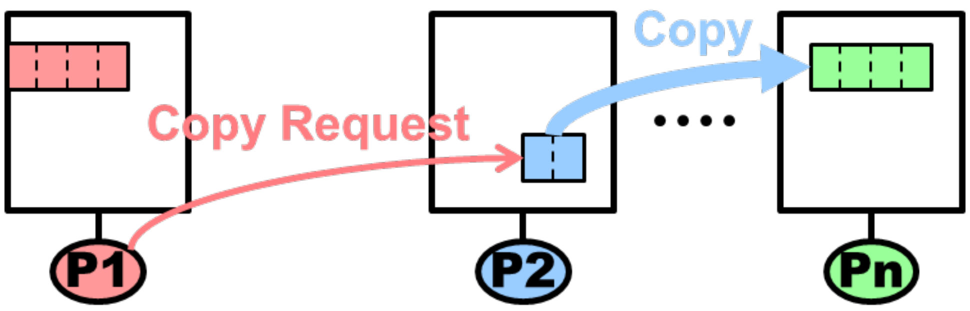
Any local memory space of any process can be mapped to this space via the registration function, **`acp_register_memory`**, of this layer. It returns a key with the data type of **`acp_atkey_t`**. There is also a de-registration function, **`acp_unregister_memory`**, to unregister the region of the specified key. Global address in a registered region can be retrieved by a query function, **`acp_query_ga`**. It returns the address with the data type of **`acp_ga_t`**.

At the registration, in addition to the start address and the size of the local region to be mapped, the color of the global address can be specified. Color is used for specifying device in the node to be used for making remote access to the region. There is a function, **`acp_colors`**, to query the maximum number of colors available on the current system.

At the initialization, a special region, called starter memory, is allocated on each process. Each of the starter memories of the processes is registered to the global address space, and there is a function, **`acp_query_starter_ga`**, to query for the global address of the starter memory of the specified rank. This region is mainly used for exchanging global addresses before performing RDMA operations on the global address space.

2.2.7 Global Memory Access

The RDMA operations on the global memory space of the basic layer is called global memory access (GMA). ACP prepares GMA functions, such as **`acp_copy`**, **`acp_add4`**, **`acp_add8`**, **`acp_cas4`** and so on, to perform copy and atomic operations on the global memory space. Unlike other communication libraries, both of the source and the destination of copy function can be remote.



All of the GMA functions are non-blocking. Therefore, each of them returns a GMA handle with the data type of **acp_handle_t** to wait for the completion. Each GMA function has an argument 'order' to specify the condition for starting the operation. If a GMA handle is specified for this argument, it will wait for the completion of the GMA of the handle before starting its access. If **ACP_HANDLE_NULL** is specified as the order, it starts the access immediately. If **ACP_HANDLE_ALL** is specified, it will wait for the completions of all of the previously invoked GMAs. If **ACP_HANDLE_CONT** is specified as the order, and if the GMA that is invoked immediately before this one has the same source rank, target rank, source color and target color, it can start its access with the same condition of the previous one, as far as it will not overtake the order. If there is any difference in those parameters, the behavior is same as the case with **ACP_HANDLE_ALL**. There can be an implementation that always performs the same way as the case with **ACP_HANDLE_ALL**, even if **ACP_HANDLE_CONT** is specified.

The completion function, **acp_complete**, waits for the completion of the GMA specified by the handle. The completions of GMA functions are ensured in-order. This means that the completion of one GMA function ensures the completion of all of the other GMA functions that have been invoked earlier than it on the process. There is also a function, **acp_inquire**, which check the completions of the GMA specified by the handle and the GMAs that have been invoked before it.

2.3 Usage

2.3.1 Installation

The simplest way to install ACP library is typically a combination of running "configure" and "make". Execute the following commands to install the ACP library system from within the directory at the top of the tree:

```
shell$ ./configure --prefix=/where/to/install
 [...lots of output...]
shell$ make all install
```

If you need special access to install, then you can execute "make all" as a user with write permissions in the build tree, and a separate "make install" as a user with write permissions to the install tree.
See INSTALL for more details.

2.3.2 Compilation

ACP provides "wrapper" compilers that should be used for compiling ACP applications:

C:	`acpcc`, `acpgcc`
C++:	`acpc++`, `acpcxx`
Fortran:	`acpfc` (not provided yet)

For example:

```
shell$ acpcc hello_world_acp.c -o hello_world_acp -g
shell$
```

All the wrapper compilers do is add a variety of compiler and linker flags to the command line and then invoke a back-end compiler. To be specific: the wrapper compilers do not parse source code at all; they are solely command-line manipulators, and have nothing to do with the actual compilation or linking of programs. The end result is an ACP executable that is properly linked to all the relevant libraries.
Customizing the behavior of the wrapper compilers is possible (e.g., changing the compiler [not recommended] or specifying additional compiler/linker flags).

2.3.3 Execution

ACP library supports acprun program to launch ACP appliations. For example:

```
  shell$ acprun -np 2 hello_world_acp   .
```

This launches two `hello_world_acp` applications on localhost. Option "`-np 2`" is same as long option "`--acp-nprocs=2`". Please note that "`-np nprocs`" has to be 'first' option of `acprun` command. There are no such limitations for the long option.
You can specify a `--acp-nodefile=nodefile` option to indicate hostnames on which `hello_world_acp` command will be launched.

If you want launch two processes on pc01 and pc02, use following command:

```
  shell$ acprun -np 2 --acp-nodefile=nodefile hello_world_acp
```

with the following nodefile:

```
---------------------------------------------------------------------------
pc01
pc01
pc02
pc02
---------------------------------------------------------------------------
```

To specify network device, for example infiniband (or udp, tofu), use `--acp-envent=ib` (or udp, tofu) option as follows:

```
  shell$ acprun -np 2 --acp-nodefile=nodefile --acp-envnet=ib
hello_world_acp
```

If you want control starter memory size of ACP basic layer library, use `--acp-startermemsize=size` (byte unit) option as follows:

```
  shell$ acprun -np 2 --acp-nodefile=nodefile --acp-envnet=ib
--acp-startermemsize=4096 hello_world_acp
```

Notice:

1. Other "`--acp-*`" options are parsed as errors in regard to invalid option, even though that option is used as user program option.
2. Tofu network device may not supported in this version.

3. API

3.1 Infrastructure

The infrastructure interface provides fundamental functions that are commonly used among communication models of ACP.

3.1.1 acp_init

int acp_init (int * *argc*, char * *argv*)**

ACP initialization.
Initializes the ACP library. Must be invoked before other functions of ACP. argc and argv are the pointers for the arguments of the main function. It returns after all of the processes complete initialization. In this function, the initialization functions of the modules of the middle layer are invoked.

Parameters:

argc	A pointer for the number of arguments of the main function.
argv	A pointer for the array of arguments of the main function.

Return values:

0	Success
-1	Fail

3.1.2 acp_finalize

int acp_finalize (void)

ACP finalization.
Finalizes the ACP library. All of the resources allocated in the library before this function are freed. It returns after all of the processes complete finalization. In this function, the finalization functions of the modules of the middle layer are invoked.

Return values:

0	Success
-1	Fail

3.1.3 acp_abort

`void acp_abort (const char * str)`

ACP abort.
Aborts the ACP library. It prints out the error message specified as the argument and the system message according to the error number ACP_ERRNO. ACP_ERRNO holds a number that shows the reason of the fail of the functions of ACP basic layer.

Parameters::

str Additional error message.

3.1.4 acp_procs

`int acp_procs (void)`

Query for the number of processes.
Returns the number of the processes.

Return values:

>=1 Number of processes.

-1 Fail

3.1.5 acp_rank

`int acp_rank (void)`

Query for the process rank.
Returns the rank number of the process that called this function.

Return values:

>=0 Rank number of the process.

-1 Fail

3.1.6 acp_sync

`int acp_sync (void)`

ACP Syncronization.
Synchronizes among all of the processes. Returns after all of the processes call this function.

Return values:

0	Success
-1	Fail

3.1.7 acp_reset

`int acp_reset (int` ***`rank`*****`)`**

ACP Re-initialization.
Re-initializes the ACP library. As rank, the new rank number of this process after this function is specified. All of the resources allocated in the library before this function are freed. The starter memory of each process is cleared to be zero. This function returns after all of the processes complete re-initialization. In this function, the functions for the initialization and the finalization of the modules of the middle layer are invoked.

Parameters:

rank	New rank number of this process after re-initialization.

Return values:

0	Success
-1	Fail

3.2 Channel

The channel interface provides functions for transferring messages via channels that are established between specific pairs of processes.

3.2.1 Data types and macros

Data types:

`acp_ch_t`

`acp_request_t`

Macros:

`ACP_CH_NULL`	Represents no channel.
`ACP_REQUEST_NULL`	Represents fail of the non-blocking operation.

3.2.2 acp_create_ch

`acp_ch_t acp_create_ch (int` ***`sender`*****`,  int`** ***`receiver`*****`)`**

Creates an endpoint of a channel to transfer messages from sender to receiver.
Creates an endpoint of a channel to transfer messages from sender to receiver, and returns a handle of it. It returns error if sender and receiver is same, or the caller process is neither the sender nor the receiver. This function does not wait for the completion of the connection between sender and receiver. The connection will be completed by the completion of the communication through this channel. There can be more than one channels for the same sender-receiver pair.

Parameters:

sender	Rank of the sender process of the channel.
receiver	Rank of the receiver process of the channel.

Return values:

ACP_CH_NULL	Fail
otherwise	A handle of the endpoint of the channel.

3.2.3 acp_free_ch

`int acp_free_ch (acp_ch_t ch)`

Frees the endpoint of the channel specified by the handle.
Frees the endpoint of the channel specified by the handle. It waits for the completion of negotiation with the counter peer of the channel for disconnection. It returns error if the caller process is neither the sender nor the receiver. Behavior of the communication with the handle of the channel endpoint that has already been freed is undefined.

Parameters:

ch	Handle of the channel endpoint to be freed.

Return values:

0	Success
-1	Fail

3.2.4 acp_nbfree_ch

`acp_request_t acp_nbfree_ch (acp_ch_t ch)`

Starts a non-blocking free of the endpoint of the channel specified by the handle.
It returns error if the caller process is neither the sender nor the receiver. Otherwise, it returns a handle of the request for waiting the completion of the free operation. Communication with the handle of the channel endpoint that has been started to be freed causes an error.

Parameters:

ch	Handle of the channel endpoint to be freed.

Return values:

ACP_REQUEST_NULL	Fail
otherwise	A handle of the request for waiting the completion of this operation.

3.2.5 acp_nbsend_ch

acp_request_t acp_nbsend_ch (acp_ch_t *ch*, void * *buf*, size_t *size*)

Non-Blocking send via channels.
Starts a non-blocking send of a message through the channel specified by the handle. It returns error if the sender of the channel endpoint specified by the handle is not the caller process. Otherwise, it returns a handle of the request for waiting the completion of the non-blocking send.

Parameters:

ch	Handle of the channel endpoint to send a message.
buf	Initial address of the send buffer.
size	Size in byte of the message.

Return values:

ACP_REQUEST_NULL	Fail.
otherwise	A handle of the request for waiting the completion of this operation.

3.2.6 acp_nbrecv_ch

acp_request_t acp_nbrecv_ch (acp_ch_t *ch*, void * *buf*, size_t *size*)

Non-Blocking receive via channels.
Starts a non-blocking receive of a message through the channel specified by the handle. It returns error if the receiver of the channel endpoint specified by the handle is not the caller process. Otherwise, it returns a handle of the request for waiting the completion of the non-blocking receive. If the message is smaller than the size of the receive buffer, only the region of the message size, starting from the initial address of the receive buffer is modified. If the message is larger than the size of the receive buffer, the exceeded part of the message is discarded.

Parameters:

ch Handle of the channel endpoint to receive a message.
buf Initial address of the receive buffer.
size Size in byte of the receive buffer.

Return values:

ACP_REQUEST_NULL Fail.
otherwise A handle of the request for waiting the completion of this operation.

3.2.7 acp_wait_ch

`size_t acp_wait_ch (acp_request_t` *`request`*`)`

Waits for the completion of the non-blocking operation.
Waits for the completion of the non-blocking operation specified by the request handle. If the operation is a non-blocking receive, it returns the size of the received data.

Parameters:

request Handle of the request of a non-blocking operation.

Return values:

>=0 Success. if the operation is a non-blocking receive, the size of the received data.
-1 Fail.

3.3 Vector

The vector interface provides functions to manipulate dynamic array. Currently the set of functions is incomplete. More functions are planned to be added.

3.3.1 Data types and macros

Data types:

```
typedef acp_ga_t acp_vector_t
typedef int acp_vector_it_t
```

Macros:

`ACP_VECTOR_NULL`	Represents no vector data.

3.3.2 acp_create_vector

`acp_vector_t acp_create_vector (size_t` *`nelem`*`, size_t` *`size`*`, int` *`rank`*`)`

Vector creation.

Creates a vector type data on any process.

Parameters:

nelem	Number of elements.
size	Size of element
rank	Rank number

Return values:

ACP_VECTOR_NULL	Fail
otherwise	A reference of created vector data.

3.3.3 acp_destroy_vector

`void acp_destroy_vector (acp_vector_t` *`vector`*`)`

Vector destruction.

Destroys a vector type data.

Parameters:

vector	A reference of vector data.

3.3.4 acp_duplicate_vector

acp_vector_t acp_duplicate_vector (acp_vector_t *vector*, int *rank*)

Vector duplicate.

Duplicate a specified vector type data on any processes.

Parameters:

vector A reference of vector data to duplicate.

rank A rank number of the process on which a vector type data is duplicated.

Return values:

ACP_VECTOR_NULL Fail

otherwise A reference of duplicated vector data.

3.3.5 acp_swap_vector

void acp_swap_vector (acp_vector_t *v1*, acp_vector_t *v2)*

Swap elements between two vectors.

Parameters:

v1 A reference of vector data to be swapped.

v2 Another reference of vector data to be swapped.

3.4 List

The list interface provides functions for establishing the data structure and algorithms of bidirectional linked list. Currently the set of functions is incomplete. More functions are planned to be added.

3.4.1 Data types and macros

Data types:

```
typedef acp_ga_t acp_list_t
typedef int acp_list_it_t
```

Macros:

`ACP_LIST_NULL`	Represents no list data.

3.4.2 acp_create_list

`acp_list_t acp_create_list (size_t` ***`elsize`*****`,   int`** ***`rank`*****`)`**

List creation.

Creates a list type data on any process.

Parameters:

elsize	Size of element.
rank	Rank number.

Return values:

ACP_LIST_NULL	Fail
otherwise	A reference of created list data.

3.4.3 acp_destroy_list

`void acp_destroy_list (acp_list_t` ***`list`*****`)`**

Destroys a list type data.

Parameters:

list	A reference of list data.

3.4.4 acp_insert_list

`acp_list_it_t acp_insert_list (acp_list_t` *`list`*`, acp_list_it_t` *`it`*`, void *` *`ptr`*`, int` *`rank`*`)`

Copy an element to the specified process and inserts it into the specified position of the list.

Parameters:

list A reference of list type data.

it An iterater of list type data.

ptr The starting address of list element.

rank Rank of the process in which the element is copied.

Return values:

it The iterator that points to the inserted element.

3.4.5 acp_erase_list

`acp_list_it_t acp_erase_list (acp_list_t` *`list`*`, acp_list_it_t` *`it`*`)`

Erase a list element.

Parameters:

list A reference of list type data.

it An iterator of list type data.

Return values:

it The iterator that points to the element which is immediately after the erased one.

3.4.6 acp_push_back_list

`void acp_push_back_list (acp_list_t` ***`list`*****`, void * `*****`ptr`*****`, int `*****`rank`*****`)`**

Erase a list element.

Parameters:

list A reference of list type data.

ptr A pointer of list type data.

rank Rank of the process in which the element is copied.

3.4.7 acp_begin_list

`acp_list_it_t acp_begin_list (acp_list_t` ***`list`*****`)`**

Query for the head iterater of a list.

Parameters:

list A reference of list type data.

Return values:

it The head iterator of the list.

3.4.8 acp_end_list

`acp_list_it_t acp_end_list (acp_list_t` ***`list`*****`)`**

Query for the tail iterator of a list.

Return values:

it The tail iterator of the list.

Parameters:

list A reference of list type data.

3.4.9 acp_increment_list

void acp_increment_list (acp_list_it_t * *list*)

Increments the iterator of a list data.

Parameters:

list A reference of list type data.

3.4.10 acp_decrement_list

void acp_decrement_list (acp_list_it_t * *list*)

Decrements the iterator of a list data.

Parameters:

list A reference of list type data.

3.5 Malloc

The malloc interface provides functions for asynchronous allocation and de-allocation of global memory.

3.5.1 acp_malloc

`acp_ga_t acp_malloc (size_t, int)`

3.5.2 acp_free

`void acp_free (acp_ga_t)`

3.6 Global Memory Management

The global memory management interface provides functions for manipulating memory regions of the global address space.

3.6.1 Data types and macros

Data types:

`typedef uint64_t acp_atkey_t`

Key for address translation of the registered memory region.

`typedef uint64_t acp_ga_t`

Global address. Byte-wise address unique among all of the processes.

Macros:

`ACP_ATKEY_NULL` Represents no address translation key.

`ACP_GA_NULL` Represents no global address.

3.6.2 acp_register_memory

`acp_atkey_t acp_register_memory (void *` ***`addr`*****`, size_t`** ***`size`*****`, int`** ***`color`*****`)`**

Memory registration.

Registers the specified memory region to global memory and returns an address translation key for it. The color that will be used for GMA with the address is also included in the key.

Parameters:

addr Logical address of the top of the memory region to be registered.

size Size of the region to be registered.

color Color number that will be used for GMA with the global memory.

Return values:

ACP_ATKEY_NULL Fail

otherwise Address translation key.

3.6.3 acp_unregister_memory

`int acp_unregister_memory (acp_atkey_t atkey)`

Memory unregistration.
Unregister the memory region with the specified address translation key.

Parameters:

atkey	Address translation key.

Return values:

0	Success
-1	Fail

3.6.4 acp_query_ga

`acp_ga_t acp_query_ga (acp_atkey_t` ***`atkey`*****`,  void *`** ***`addr`*****`)`**

Query for the global address.
Returns the global address of the specified logical address translated by the specified address translation key.

Parameters:

atkey	Address translation key.
addr	Logical address.

Return values:

ACP_GA_NULL	Fail
otherwise	Global address of starter memory.

3.6.5 acp_colors

`int acp_colors (void)`

Query for the maximum number of colors.
Returns the maximum number of colors on this environment.

Return values:

>=1	Maximum number of colors.
-1	Fail

3.6.6 acp_query_rank

`int acp_query_rank (acp_ga_t` *`ga`*`)`

Query for the rank of the global address.
Returns the rank of the process that keeps the logical region of the specified global address. It can be used for retrieving the rank of the starter memory. It returns -1 if the ACP_GA_NULL is specified as the global address.

Parameters:

ga	Global address.

Return values:

>=0	Rank number.
-1	Fail

3.6.7 acp_query_color

`int acp_query_color (acp_ga_t *ga*)`

Query for the color of the global address.
Returns the color of the specified global address. It returns -1 if the ACP_GA_NULL is specified as the global address.

Parameters:

ga Global address.

Return values:

>=0 Color number.

-1 Fail

3.6.8 acp_query_address

`void* acp_query_address (acp_ga_t *ga*)`

Query for the logical address.
Returns the logical address of the specified global address. It fails if the process that keeps the logical region of the global address is different from the caller. It can be used for retrieving logical address of the starter memory.

Parameters:

ga Global address.

Return values:

NULL FailLogical address.

otherwise Logical address.

3.6.9 acp_query_starter_ga

`acp_ga_t acp_query_starter_ga (int` *`rank`*`)`

Query for the global address of the starter memory.
Returns the global address of the starter memory of the specified rank.

Parameters:

rank	Rank number.

Return values:

ACP_GA_NULL	Fail	Global address of the starter memory.
otherwise	Global address of the starter memory.	

3.7 Global Memory Access

The global memory access interface provides functions for RDMA operations on the global address space.

3.7.1 Data types and macros

Data types:

`typedef uint64_t acp_handle_t`

Handle of GMA. Used to distinguish incomplete GMA.

Macros:

`ACP_HANDLE_ALL` Represents handles of all advancing GMAs.

`ACP_HANDLE_CONT` Represents continuation of the GMA invoked just before this one.

`ACP_HANDLE_NULL` Represents no GMA handle.

If ACP_HANDLE_CONT is specified as the order of a GMA, and if the GMA that is invoked immediately before this one has the same source rank, target rank, source color and target color, it can start its access with the same condition of the previous one, as far as it will not overtake the order. If there is any difference in those parameters, the behavior is same as the case with ACP_HANDLE_ALL. There can be an implementation that always performs the same way as the case with ACP_HANDLE_ALL, even if ACP_HANDLE_CONT is specified.

3.7.2 acp_copy

`acp_handle_t acp_copy (acp_ga_t *dst*, acp_ga_t *src*, size_t *size*, acp_handle_t *order*)`

Copy.
Copies data of the specified size between the specified global addresses of the global memory. Ranks of both of dst and src can be different from the rank of the caller process.

Parameters:

dst	Global address of the head of the destination region of the copy.
src	Global address of the head of the source region of the copy.
size	Size of the data to be copied.
order	The handle to be used as a condition for starting this GMA.

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.3 acp_add4

acp_handle_t acp_add4 (acp_ga_t *dst*, acp_ga_t *src*, uint32_t *value*, acp_handle_t *order*)

4byte Add

Performs an atomic add operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be added is 4byte. Global addresses must be 4byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
value	Value to be added.
order	The handle to be used as a condition for starting this GMA.

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.4 acp_add8

acp_handle_t acp_add8 (acp_ga_t *dst*, acp_ga_t *src*, uint64_t *value*, acp_handle_t order)

8byte Add
Performs an atomic add operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be added is 8byte. Global addresses must be 8byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
value	Value to be added.
order	The handle to be used as a condition for starting this GMA.

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.5 acp_cas4

acp_handle_t acp_cas4 (acp_ga_t *dst*, acp_ga_t *src*, uint32_t *oldval*, uint32_t *newval*, acp_handle_t *order*)

4byte Compare and Swap
Performs an atomic compare-and-swap operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be compared and swapped is 4byte. Global addresses must be 4byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
oldval	Old value to be compared.
newval	New value to be swapped.
order	The handle to be used as a condition for starting this GMA.

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.6 acp_cas8

`acp_handle_t acp_cas8 (acp_ga_t`*`dst`*`, acp_ga_t`*`src`*`, uint64_t`*`oldval`*`, uint64_t`*`newval`*`, acp_handle_t`*`order`*`)`

8byte Compare and Swap

Performs an atomic compare-and-swap operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be compared and swapped is 8byte. Global addresses must be 8byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
oldval	Old value to be compared.
newval	New value to be swapped.
order	The handle to be used as a condition for starting this GMA.

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.7 acp_and4

acp_handle_t acp_and4 (acp_ga_t *dst*, acp_ga_t *src*, uint32_t value, acp_handle_t *order*)

4byte AND

Performs an atomic AND operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be applied is 4byte. Global addresses must be 4byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
value	Value to be applied the AND operation.
order	The handle to be used as a condition for starting this GMA.

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.8 acp_and8

acp_handle_t acp_and8 (acp_ga_t *dst*, acp_ga_t *src*, uint64_t *value*, acp_handle_t *order*)

8byte AND

Performs an atomic AND operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be applied is 8byte. Global addresses must be 8byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
value	Value to be applied the AND operation.
order	The handle to be used as a condition for starting this GMA.

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.9 acp_or4

acp_handle_t acp_or4 (acp_ga_t *dst*, acp_ga_t *src*, uint32_t *value*, acp_handle_t *order*)

4byte OR

Performs an atomic OR operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be applied is 4byte. Global addresses must be 4byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
value	Value to be applied the OR operation.
order	The handle to be used as a condition for starting this GMA.

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.10 acp_or8

`acp_handle_t acp_or8 (acp_ga_t`*`dst`*`, acp_ga_t`*`src`*`, uint64_t`*`value`*`, acp_handle_t`*`order`*`)`

8byte OR

Performs an atomic OR operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be applied is 8byte. Global addresses must be 8byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
value	Value to be applied the OR operation.
order	The handle to be used as a condition for starting this GMA.

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.11 acp_xor4

`acp_handle_t acp_xor4 (acp_ga_t` *`dst`*`, acp_ga_t` *`src`*`, uint32_t` *`value`*`, acp_handle_t` *`order`*`)`

4byte Exclusive OR
Performs an atomic XOR operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be applied is 4byte. Global addresses must be 4byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
value	Value to be applied the XOR operation.
order	The handle to be used as a condition for starting this GMA

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.12 acp_xor8

`acp_handle_t acp_xor8 (acp_ga_t` *`dst`*`, acp_ga_t` *`src`*`, uint64_t` *`value`*`, acp_handle_t` *`order`*`)`

Performs an atomic XOR operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be applied is 8byte. Global addresses must be 8byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
value	Value to be applied the XOR operation.
order	The handle to be used as a condition for starting this GMA

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.13 acp_swap4

`acp_handle_t acp_swap4 (acp_ga_t` ***`dst`*****`,  acp_ga_t`** ***`src`*****`,  uint32_t`** ***`value`*****`,`**
`acp_handle_t` ***`order`*****`)`**

4byte Swap
Performs an atomic swap operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be swapped is 4byte. Global addresses must be 4byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation
value	Value to be swapped.
order	The handle to be used as a condition for starting this GMA

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.14 acp_swap8

`acp_handle_t acp_swap8 (acp_ga_t`*`dst`*`, acp_ga_t`*`src`*`, uint64_t`*`value`*`, acp_handle_t`*`order`*`)`

8byte Swap

Performs an atomic swap operation on the global address specified as src. The result of the operation is stored in the global address specified as dst. The rank of the dst must be the rank of the caller process. The values to be swapped is 8byte. Global addresses must be 8byte aligned.

Parameters:

dst	Global address to store the result.
src	Global address to apply the operation.
value	Value to be swapped.
order	The handle to be used as a condition for starting this GMA.

Return values:

ACP_HANDLE_NULL	Fail
otherwise	A handle for this GMA.

3.7.15 acp_complete

`void acp_complete (acp_handle_t`*`handle`*`)`

Completion of GMA.

Complete GMAs in order. It waits until the GMA of the specified handle completes. This means all the GMAs invoked before that one are also completed. If ACP_HANDLE_ALL is specified, it completes all of the out-standing GMAs. If the specified handle is ACP_HANDLE_NULL, the handle of the GMA that has already been completed, or the handle of the GMA that has not been invoked, this function returns immediately.

Parameters:

handle	Handle of a GMA to be waited for the completion.

3.7.16 acp_inquire

`int acp_inquire (acp_handle_t *handle*)`

Query for the completion of GMA.
Queries if any of the GMAs that are invoked earlier than the GMA of the specified handle, including that GMA, are incomplete. It returns zero if all of those GMAs have been completed. Otherwise, it returns one. If ACP_HANDLE_ALL is specified, it checks of the out-standing GMAs. If the specified handle is ACP_HANDLE_NULL, the handle of the GMA that has already been completed, or the handle of the GMA that has not been invoked, it returns zero.

Parameters:

handle	Handle of the GMA to be checked for the completion.

Return values:

0	No incomlete GMAs.
1	There is at least one incomplete GMA.