Specification of Advanced Communication Primitives (ACP) Library

Version 2.0 (draft)

13 June, 2016

Acknowledgements

Organizations and Members of ACE Project:

- Kyushu University
  Takeshi Nanri, 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

- Oita University
  Toshiya Takami

Updates

From version 1.2 to 2.0

- New facilities
  1. Added following commands to launch multiple MPI programs with ACP:
     macprun

- New interfaces
  1. Added functions of the following data structures:
     Vector, Deque and List

- Modified interfaces
  1. Definitions of the following functions have been changed:
     acp_insert_list, acp_push_back_list.

- Fixed bugs and problems
  1. acpbl_ib.c: Solved following issues:
     - performance of acp_sync()
     - CPU utilization in the communication thread
     - memory usage

- Known issues
  1. Facility for launching multiple MPI programs with ACP is not supported on Tofu interconnect, yet.

From version 1.1 to 1.2

- New interfaces
  1. Added following functions:
     acp_clear_vector, acp_end_map

- Modified interfaces
  1. Definitions of the following functions have been changed:

acp_insert_list, acp_push_back_list, acp_insert_map, acp_find_map.
2. Names and definitions of some functions have been changed as follows:
   - old: acp_increment_list, new: acp_increment_list_it
   - old: acp_decrement_list, new: acp_decrement_list_it

- Fixed bugs and problems
  1. acpcl.c: Fixed bugs that had caused hangs in some cases. Also, changed the protocol for long messages from rendezvous to pipelined-eager.
  2. acpbl_ib.c: Fixed bugs that had caused incorrect behaviors in acp_init, acp_colors, acp_register_memory and acp_unregister_memory.
  3. acpdl_malloc.c: Modified algorithms for acp_malloc and acp_free to achieve better performance.

- Known issues
  1. :
     - .

From version 1.0 to 1.1

- New interfaces
  1. Map interface

- Fixed bugs
  1. acpcc: Fixed behavior with -l option.
  2. acpcl.c: Fixed to enable thread-safety.
  3. acpcl.c, acpcl_progress.{c,h}: Fixed the way to notify the arrivals of connection requests.

- Known issues
  2. acpbl_ib.c:
     - In some cases, arrivals of operations for remote atomic and remote-remote copy may be handled incorrectly.
     - Performance drops significantly when large number of operations are invoked.

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.

sender
channel
receiver
send
receive

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 constructor function, **acp_create_vector**. As an argument, this function accepts the rank to place the instance of the vector.

2.2.4   Deque

The deque interface provides the data structure and algorithms of double ended queue. The type of for referencing deques is **acp_deque_t**, and the type of the iterator of deques is **acp_deque_it_t**. Deques can be created via the constructor function, **acp_create_deque**. As an argument, this function accepts the rank to place the instance of the deque.

2.2.5   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.

2.2.6   Map

The map interface provides the data structure and algorithms of associative arrays. The type for referencing maps is **acp_map_t**, the type for the iterator of maps is **acp_map_it_t**, and the type for returning result of finding element in a map is **acp_map_ib_t**. Maps are created via the constructor function, **acp_create_map**. This function prepares an empty map distributed among a group of processes. As an argument, it accepts the number of processes, the array of ranks of the processes in the group, the number of slots and the rank to place the control structure of the map. Elements can be added to a map via the function **acp_insert_map**. In addition to the map to insert, this function accepts the key and the value of the element, as an argument. Each of the key and the value are specified by a pair of the address and the size in byte. The rank to place the element to be inserted is decided according to the hash value calculated from the key. **acp_find_map** searches the key in a map. It returns the value true or false according to the result, and the iterator of the element with the key if it has been found.

2.2.7   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.

2.2.8   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.

2.2.9 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.
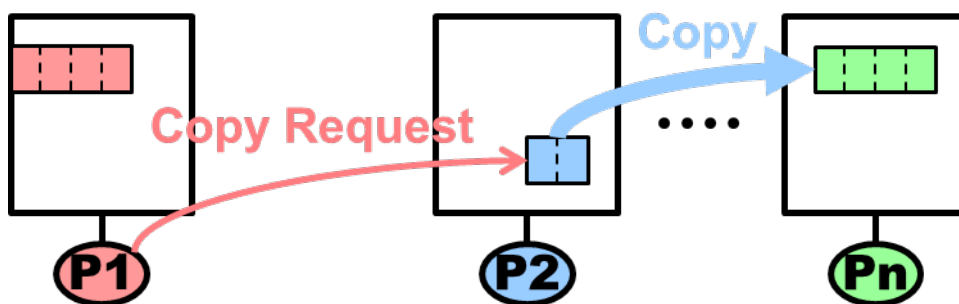
All of the GMA functions are non-blocking. Therefore, each of them returns a GMA handle with the data type of `acp_handle_t` to wait for the completion. Each GMA function has an argument 'order' to specify the condition for starting the operation. If a GMA handle is specified for this argument, it will 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.

3. Usage

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.

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).

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 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-nodefile=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-nodefile=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.

4. API

4.1 Infrastructure

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

4.1.1 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

4.1.2 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

4.1.3 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.

4.1.4 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

4.1.5 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

4.1.6 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

4.1.7 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

4.2 Channel

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

4.2.1 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.

4.2.2 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.

4.2.3 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

4.2.4 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.

4.2.5   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.

4.2.6 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.

4.2.7 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.

4.3 Vector

The vector interface provides functions to manipulate dynamic array. Each element of a vector can be manipulated by an iterator. An iterator is valid only at the process that retrieved it. Iterators will be invalid after modification of the vector by: **acp_assign_vector**, **acp_assign_range_vector**, **acp_clear_vector**, **acp_destroy_vector**, **acp_erase_vector**, **acp_erase_range_vector**, **acp_insert_vector**, **acp_insert_range_vector** and **acp_swap_vector**.

4.3.1 Data types and macros

Data types:

```
typedef struct {
        acp_ga_t ga;
} acp_vector_t
typedef struct {
        acp_vector_t vector;
        int index;
} acp_vector_it_t
```

4.3.2 acp_assign_vector

```
void acp_assign_vector(acp_vector_t vector1,acp_vector_t vector2);
```

Copy data from *vector2* to *vector1*. Old data in *vector1* is overwritten.

Parameters:

      vector1   Destination vector.

      vector2   Source vector.

4.3.3 acp_assign_vector

```
void acp_assign_range_vector(acp_vector_t vector,
acp_vector_it_t start, acp_vector_it_t end);
```

Copy a range of data in a **vector** to another **vector**. The iterators of the **start** and the **end** of the range must be in the same **vector**. Old data in the destination **vector** is overwritten.

Parameters:

  vector Destination vector.

  start Iterator that specifies the start of the range of the vector to be copied.

  end Iterator that specifies the end of the range of the vector to be copied.

4.3.4 acp_at_vector

```
acp_ga_t acp_at_vector(acp_vector_t vector, int offset);
```

Query for the global address of the specified position in a **vector.**

Parameters:

  vector Vector.

  offset Offset from the beginning of the vector.

Return values:

  ACP_GA_NULL Fail

  otherwise Global address of the specified position.

4.3.5 acp_begin_vector

```
acp_vector_it_t acp_begin_vector(acp_vector_t vector);
```

Query for the head iterator of a **vector.**

Parameters:

  vector Vector.

Return values:

  The head iterator of the vector.

4.3.6   acp_capacity_vector

```
size_t acp_capacity_vector(acp_vector_t vector);
```

Query for the capacity of the **vector** without extending area.

Parameters:

> **vector**   Vector.

Return values

> The capacity of the vector without extending area.

4.3.7   acp_clear_vector

```
void acp_clear_vector(acp_vector_t vector);
```

Set the size of **vector** to be zero.

Parameters:

> **vector**   Vector.

4.3.8   acp_create_vector

```
acp_vector_t acp_create_vector(size_t size, int rank);
```

Creates a vector on the specified process.

Parameters:

> **size**   Size of the vector
>
> **rank**   Rank of the process on which the vector is located

Return values;

> **ga == ACP_GA_NULL**   Fail
>
> *otherwise*   **Created vector**

4.3.9 acp_destroy_vector

```
void acp_destroy_vector (acp_vector_t vector);
```

Destroy *vector*.

Parameters:

     vector   Vector.

4.3.10 acp_empty_vector

```
int acp_empty_vector(acp_vector_t vector);
```

Query for the emptiness of *vector*.

Parameters:

     vector   Vector.

Return values;

     1   Empty.

     0   Not empty.

4.3.11 acp_end_vector

```
acp_vector_it_t acp_end_vector(acp_vector_t vector);
```

Query for the iterator just after the tail element of *vector*.

Parameters:

     vector   Vector.

Return values;

     The iterator just after the tail element of the vector.

4.3.12 acp_erase_vector

```
acp_vector_it_t acp_erase_vector(acp_vector_it_t it, size_t size);
```

Erase data from a vector.

Parameters:

it Iterator that points to the beginning of the range to be erased.

size Size of the data to be erased.

Return values;

The iterator of the data immediately after the erased data.

4.3.13 acp_erase_range_vector

```
acp_vector_it_t acp_erase_range_vector(acp_vector_it_t start,
 acp_vector_it_t end);
```

Erase a range of data from a vector.

Parameters:

start Iterator that points to the beginning of the range to be erased.

end Iterator that points to the end of the range to be erased

Return values;

The iterator of the data immediately after the erased data

4.3.14 acp_insert_vector

```
acp_vector_it_t acp_insert_vector(acp_vector_it_t it,
const acp_ga_t ga, size_t size);
```

Inserts data into the specified position of the vector.

Parameters:

it   Iterator of a vector that specifies the position to insert data.

ga   Global address of the data to be inserted.

size   Size of the data to be inserted.

Return values:

The iterator that points to the inserted data.

4.3.15 acp_insert_range_vector

```
acp_vector_it_t acp_insert_range_vector(acp_vector_it_t it,
acp_vector_it_t start, acp_vector_it_t end);
```

Insert range of data from a vector to another vector.

Parameters:

it   Iterator of a vector that specifies the position to insert data.

start   Iterator of a vector that specifies the beginning position of the data to be inserted.

end   Iterator of a vector that specifies the ending position of the data to be inserted.

Return values:

The iterator that points to the inserted data.

4.3.16 acp_pop_back_vector

```
void acp_pop_back_vector(acp_vector_t vector, size_t size);
```

Erase specified size of data from the tail of *vector*.

Parameters:

> *vector* Vector.
>
> *size* Size of the data to erase.

4.3.17 acp_push_back_vector

```
void acp_push_back_vector(acp_vector_t vector,
const acp_ga_t ga, size_t size);
```

Add specified size of data to the tail of *vector*.

Parameters:

> *vector* Vector.
>
> *ga* Global address of the data to be added.
>
> *size* Size of the data to be added.

4.3.18 acp_reserve_vector

```
void acp_reserve_vector(acp_vector_t vector, size_t size);
```

Change the size of *vector* so that it can store the specified number of elements.

Parameters:

> *vector* Vector.
>
> *size* New number of elements of the vector.

4.3.19 acp_size_vector

`size_t acp_size_vector(acp_vector_t vector);`

Query for the size of the data stored in **vector**.

Parameters:

> **vector**   Vector.

Return values:

> The size of the data stored in the vector.

4.3.20 acp_swap_vector

`void acp_swap_vector(acp_vector_t vector1,  acp_vector_t vector2);`

Swap data between **vector1** and **vector2**.

Parameters:

> **vector1**   Vector.
>
> **vector2**   Another vector.

4.3.21 acp_advance_vector

`acp_vector_it_t acp_advance_vector_it(acp_vector_it_t it,`
`int n);`

Make advance of an iterator of a vector.

Parameters:

> **it**   Iterator of a vector.
>
> **n**   The number of advance to make. Minus number is also valid.

Return values:

> The iterator that points to the result of the advance.

4.3.22 acp_dereference_vector

```
acp_ga_t acp_dereference_vector_it(acp_vector_it_t it);
```

Query for the global address of an iterator of a vector.

Parameters:

      it   Iterator of a vector.

Return values;

      Global address of the specified iterator of the vector.

4.3.23 acp_distance_vector

```
int acp_distance_vector_it(acp_vector_it_t first,
acp_vector_it_t last);
```

Query for the distance between two iterators of a **vector.**

Parameters;

      first   Iterator of a vector.

      last   Another iterator of the vector.

Return values;

      Distance between two vectors.

4.4   Deque

The deque interface provides functions for establishing the data structure and functions of double-ended queues. Each element of a deque can be manipulated by an iterator. An iterator is valid only at the process that retrieved it. Iterators will be invalid after modification of the deque by: **acp_assign_deque, acp_assign_range_deque, acp_clear_deque, acp_destroy_deque, acp_erase_deque, acp_erase_range_deque, acp_insert_deque, acp_insert_range_deque and acp_swap_deque.**

4.4.1   Data types

Datatypes:

```
typedef struct {
    acp_ga_t ga;
} acp_deque_t;


typedef struct {
    acp_deque_t deque;
    int index;
} acp_deque_it_t;
```

4.4.2   acp_assign_deque

```
void acp_assign_deque(acp_deque_t deque1, acp_deque_t deque2);
```

Copy elements from **deque2** to **deque1.**
Old elements of **deque2** are overwritten.

Parameters:

      deque1   Destination deque

      deque2   Source deque

4.4.3 acp_assign_range_deque

```
void acp_assign_range_deque(acp_deque_t deque, acp_deque_it_t start,
acp_deque_it_t end);
```

Copy a range of elements in a *deque,* specified by *start* and *end*, to *deque*. *start* and *end* must be in the same *deque.* Old data in *deque* is overwritten.

Parameters;

      deque   Destination deque

      start   Iterator that specifies the start of the range of the deque to be copied

      end   Iterator that specifies the end of the range of the deque to be copied

4.4.4 acp_at_deque

```
acp_ga_t acp_at_deque(acp_deque_t deque, int offset);
```

Query for the global address of the position specified by *offset* in *deque.*

Parameters;

      deque   Deque

      offset   Offset from the beginning of **deque**

Return values;

      ACP_GA_NULL   Fail

      otherwise   Global address of the specified position

4.4.5 acp_back_deque

```
acp_deque_it_t acp_back_deque(acp_deque_t deque);
```

Query for the tail element of *deque.*

Parameters;

      deque   Deque

Return values;

      Iterator of the tail of **deque**

4.4.6 acp_begin_deque

`acp_deque_it_t acp_begin_deque(acp_deque_t deque);`

Query for the head element of *deque.*

Parameters;

> *deque*   Deque

Return values;

> Iterator of the head of `deque`

4.4.7 acp_capacity_deque

`size_t acp_capacity_deque(acp_deque_t deque);`

Query for the capacity of *deque* without extending area.

Parameters;

> *deque*   Deque

Return values;

> Capacity of *deque* without extending area

4.4.8 acp_clear_deque

`void acp_clear_deque(acp_deque_t deque);`

Set the size of *deque* to be zero.

Parameters;

> *deque*   Deque

4.4.9 acp_create_deque

```
acp_deque_t acp_create_deque(size_t size, int rank);
```

Creates a deque on the specified process.

Parameters;

  size   Number of elements of the deque

  rank   Rank of the process on which the deque is located

Return values;

  ga == ACP_GA_NULL   Fail

  otherwise   Created deque

4.4.10 acp_destroy_deque

```
void acp_destroy_deque(acp_deque_t deque);
```

Destroys *deque*.

Parameters;

  deque   Deque

4.4.11 acp_empty_deque

```
int acp_empty_deque(acp_deque_t deque);
```

Query for the emptiness of *deque*.

Parameters;

  deque   Deque

Return values;

  1   Empty

  0   Not empty

4.4.12   acp_end_deque

`acp_deque_it_t acp_end_deque(acp_deque_t deque);`

Query for the iterator just after the tail element of *deque.*

Parameters;

　　deque　Deque

Return values;

　　Iterator just after the tail element of deque.

4.4.13   acp_erase_deque

`acp_deque_it_t acp_erase_deque(acp_deque_it_t it,`
`size_t size);`

Erase range of elements from a deque.

Parameters;

　　it　　Iterator that points to the beginning of the range of the deque to be erased

　　size　Size of the range to be erased

Return values;

　　Iterator of the element of the deque immediately after the erased range

4.4.14   acp_erase_range_deque

**acp_deque_it_t   acp_erase_range_deque(acp_deque_it_t   start,
acp_deque_it_t end);**

Erase a range of elements from a deque.

Parameters;

   start   Iterator that points to the beginning of the range of the deque to be
erased

   end   Iterator that points to the end of the range of the deque to be erased

Return values;

   Iterator of the element of the deque immediately after the erased range

4.4.15   acp_front_deque

acp_deque_it_t acp_front_deque(acp_deque_t deque);

Query for the iterator just before the head element of *deque.*

Parameters;

   deque   Deque

Return values;

   Iterator just before the head element of the *deque*

4.4.16 acp_insert_deque

```
acp_deque_it_t acp_insert_deque(acp_deque_it_t it,
const acp_ga_t ga, size_t size);
```

Insert elements into the specified position of another deque.

Parameters;

      it   Iterator of a deque that specifies the position to insert elements

      ga   Global address of the first element to be inserted

      size Number of   elements to be inserted

Return values;

      Iterator that points to the beginning of the inserted elements

4.4.17 acp_insert_range_deque

```
acp_deque_it_t acp_insert_range_deque(acp_deque_it_t it,
acp_deque_it_t start, acp_deque_it_t end);
```

Insert a range of elements in a deque to another deque.

Parameters;

      it   Iterator of a deque that specifies the position to insert elements

      start Iterator of a deque that specifies the beginning position of the range of elements to be inserted

      end   Iterator of a deque that specifies the ending position of the range of elements to be inserted

Return values;

      Iterator that points to the beginning of the inserted elements

4.4.18 acp_pop_back_deque

```
void acp_pop_back_deque(acp_deque_t deque, size_t size);
```

Erase specified number of elements from the tail of **deque.**

Parameters;

      deque   Deque

      size   Number of elements to erase

4.4.19 acp_pop_front_deque

```
void acp_pop_front_deque(acp_deque_t deque, size_t size);
```

Erase specified number of elements from the head of **deque.**

Parameters;

      deque   Deque

      size   Number of elements to erase

4.4.20 acp_push_back_deque

```
void acp_push_back_deque(acp_deque_t deque, const acp_ga_t ga,
size_t size);
```

Add specified number of elements to the tail of **deque**.

Parameters;

      deque   Deque

      ga   Global address of the first element to be inserted

      size   Number of   elements to be inserted

4.4.21　acp_push_front_deque

```
void acp_push_front_deque(acp_deque_t deque, const acp_ga_t ga,
size_t size);
```

Add specified number of elements to the head of *deque.*

Parameters;

> *deque*　　Deque
>
> *ga*　　　Global address of the first element to be inserted
>
> *size*　　Number of　elements to be inserted

4.4.22　acp_reserve_deque

```
void acp_reserve_deque(acp_deque_t deque, size_t size);
```

Change the area of *deque* so that it can store the specified number of elements.

Parameters;

> *deque*　　Deque
>
> *size*　　New number of　elements that can be stored in deque

4.4.23　acp_size_deque

```
size_t acp_size_deque(acp_deque_t deque);
```

Query for the number of elements in *deque*.

Parameters;

> *deque*　　Deque

Return values;

> Number of elements stored in *deque*

4.4.24 acp_swap_deque

```
void acp_swap_deque(acp_deque_t deque1, acp_deque_t deque2);
```

Swap elements between two **deques.**

Parameters;

> **deque1**   Deque
>
> **deque2**   Another deque

4.4.25 acp_advance_deque

```
acp_deque_it_t acp_advance_deque_it(acp_deque_it_t it, int n);
```

Make advance of an iterator of a deque.

Parameters;

> **it**   Iterator of a deque
>
> **n**   Number of elements to make advance (Minus number is also valid)

Return values;

> Iterator of the result of the advance.

4.4.26 acp_dereference_deque

```
acp_ga_t acp_dereference_deque_it(acp_deque_it_t it);
```

Query for the global address of an iterator of a deque.

Parameters;

> **it**   Iterator of a deque

Return values;

> Global address of the specified iterator

4.4.27   acp_distance_deque

```
int acp_distance_deque_it(acp_deque_it_t first, acp_deque_it_t last);
```

Query for the distance between two iterators in a deque.

Parameters;

> ***first***   Iterator of a deque
>
> ***last***   Another iterator of the deque

Return values;

> Distance between two iterators

4.5   List

The list interface provides functions for establishing the data structure and algorithms of bidirectional linked list. Each element of a list can be manipulated by an iterator. An iterator is valid only at the process that retrieved it. Iterators will be invalid after modification of the list by: **acp_assign_list, acp_assign_range_list, acp_clear_list, acp_destroy_list, acp_erase_list, acp_erase_range_list, acp_insert_list, acp_insert_range_list, acp_merge_list, acp_remove_list, acp_reverse_list, acp_splice_list, acp_swap_list and acp_unique_list**

4.5.1   Data types and macros

Data types:

```
typedef struct {
        acp_ga_t ga;
} acp_list_t
typedef struct {
        acp_list_t list;
        acp_ga_t elem;
} acp_list_it_t
```

4.5.2   acp_assign_list

```
void acp_assign_list(acp_list_t list1, acp_list_t list2);
```

Copy elements from *list2* to *list1.* Old elements of *list1* are overwritten.

Parameters:

> *list1*   Destination deque
> *list2*   Source deque

4.5.3   acp_assign_range_list

```
void acp_assign_range_list(acp_list_t list,
acp_list_it_t start, acp_list_it_t end);
```

Copy a range of elements in a list, specified by **start** and **end**, to **list**. **start** and **end** must be in the same **list**. Old data in list is overwritten. New elements are located in the same process of the source list.

Parameters:

      list   Destination list

      start Iterator that specifies the start of the range of the list to be copied

      end   Iterator that specifies the end of the range of the list to be copied

4.5.4   acp_begin_list

```
acp_list_it_t acp_begin_list(acp_list_t list);
```

Query for the head element of *list*.

Parameters:

      list   List

Return values:

      Iterator of the head of *list*

4.5.5   acp_clear_list

```
void acp_clear_list(acp_list_t list);
```

Make *list* to be empty.

Parameters:

      list   List

4.5.6 acp_create_list

`acp_list_t acp_create_list(int rank);`

Creates an empty list on the specified process.

Parameters:

   rank   Rank of the process on which the list is located

Return values:

   ga == ACP_GA_NULL   Fail

   otherwise   Created list

4.5.7 acp_destroy_list

`void acp_destroy_list(acp_list_t list);`

Destroy ***list***.

Parameters:

   list   List

4.5.8 acp_empty_list

`int acp_empty_list(acp_list_t list);`

Query for the emptiness of ***list.***

Parameters:

   list   List

Return values:

   1   Empty

   0   Not empty

4.5.9 acp_end_list

```
acp_list_it_t acp_end_list(acp_list_t list);
```

Query for the iterator just after the tail element of *list*.

Parameters:

      list List

Return values:

      Iterator just after the tail element of *list*.

4.5.10 acp_erase_list

```
acp_list_it_t acp_erase_list(acp_list_it_t it);
```

Erase an element from a list.

Parameters:

      it   Iterator that points to the beginning of the range of the list to be erased

Return values:

      Iterator of the element of the list immediately after the erased element

4.5.11 acp_erase_range_list

```
acp_list_it_t acp_erase_range_list(acp_list_it_t start,
acp_list_it_t end);
```

Erase a range of elements from a list.

Parameters:

      start Iterator that points to the beginning of the range of the list to be erased

      end   Iterator that points to the end of the range of the list to be erased

Return values:

      Iterator of the element of the list immediately after the erased range

4.5.12　acp_insert_list

acp_list_it_t acp_insert_list(acp_list_it_t it,

const acp_element_t elem, int rank);

Create an element on the specified process and insert it into the specified position of another list.

Parameters:

      it　　Iterator of a list that specifies the position to insert the element

      elem　Contents of the new element

      size　Rank of process to locate the new element

Return values:

      Iterator that points to the new element

4.5.13　acp_insert_range_list

acp_list_it_t acp_insert_range_list(acp_list_it_t it,

acp_list_it_t start, acp_list_it_t end);

Insert a range of elements in a list to another list.

Parameters:

      it　　Iterator of a list that specifies the position to insert elements

      start　Iterator of a list that specifies the beginning position of the range of elements to be inserted

      end　　Iterator of a list that specifies the ending position of the range of elements to be inserted

Return values:

      Iterator that points to the beginning of the inserted elements

4.5.14 acp_merge_list

void acp_merge_list(acp_list_t list1, acp_list_t list2,

int (*comp)(const acp_element_t elem1, const acp_element_t elem2));

Marge two sorted lists.

Parameters:

      list1   List to be marged

      list2   Another list to be marged

      comp   Pointer to a function that returns:

              minus for elem1 < elem2

              0 for elem1 == elem2

              plus for elem1 > elem2

4.5.15 acp_pop_back_list

void acp_pop_back_list(acp_list_t list);

Erase an element from the tail of *list.*

Parameters:

      list List

4.5.16 acp_pop_front_list

void acp_pop_front_list(acp_list_t list);

Erase an element from the head of *list.*

Parameters:

      list List

4.5.17   acp_push_back_list

void acp_push_back_list(acp_list_t list, const acp_element_t elem, int rank);

Create an element on the specified process and add it to the tail of *list*.

Parameters:

      list   List

      elem   Contents of the new element

      rank   Rank of process to locate the new element

4.5.18   acp_push_front_list

void acp_push_front_list(acp_list_t list, const acp_element_t elem, int rank);

Create an element on the specified process and add it to the head of *list*.

Parameters:

      list   List

      elem   Contents of the new element

      rank   Rank of process to locate the new element

4.5.19   acp_remove_list

void acp_remove_list(acp_list_t list, const acp_element_t elem);

Remove elements that has the same contents with *elem* from **list.**

Parameters:

      list   List

      elem   Contents of the element to compare

4.5.20   acp_reverse_list

void acp_reverse_list(acp_list_t list);

Reverse the order of elements of **list.**

Parameters:

      list   List

4.5.21   acp_size_list

size_t acp_size_list(acp_list_t list);

Query for the number of elements in **list.**

Parameters:

      list   List

Return values:

      Number of elements in **list**

4.5.22   acp_sort_list

void acp_sort_list(acp_list_t list,
int (*comp)(const acp_element_it elem1,const acp_element_it elem2));

Sort elements of **list.**

Parameters:

      list   List
      comp   Pointer to a function that returns:
          minus for elem1 < elem2
          0 for elem1 == elem2
          plus for elem1 > elem2

4.5.23   acp_splice_list

`void acp_splice_list(acp_list_it_t it1, acp_list_it_t is2);`

Move an element from a list to the specified place of another list.

Parameters:

it1   Iterator that points to the position to insert the element

it2   Iterator of the element to move

4.5.24   acp_splice_range_list

`void acp_splice_range_list(acp_list_it_t it, acp_list_it_t start, acp_list_it_t end);`

Move a range of   elements from a list to the specified place of another list.

Parameters:

it   Iterator that points to the position to insert the elements

start   Iterator that points to the beginning of the range to be moved

end   Iterator that points to the end of the range to be moved

4.5.25   acp_swap_list

`void acp_swap_list(acp_list_t list1, acp_list_t list2);`

Swap elements between two lists.

Parameters:

list1   List

list2   Another list

4.5.26　acp_unique_list

```
void acp_unique_list(acp_list_t list);
```

Erase redundant elements from **list.**

Parameters:

　　list　List

4.5.27　acp_advance_list

```
acp_list_it_t acp_advance_list_it(acp_list_it_t it, int n);
```

Make advance of an iterator of a list.

Parameters:

　　it　Iterator of a list

　　n　Number of elements to make advance (Minus number is also valid)

Return values:

　　Iterator of the result of the advance.

4.5.28　acp_decrement_list

```
acp_list_it_t acp_decrement_list_it(acp_list_it_t it);
```

Decrement iterator of a list.

Parameters:

　　it　Iterator of a list

Return values:

　　Previous iterator of **it**

4.5.29 acp_deregerence_list

```
acp_element_t acp_dereference_list_it(acp_list_it_t it);
```

Query for the global address and the size of an iterator of a list.

Parameters:

   it   Iterator of a list

Return values:

   Element of *it*

4.5.30 acp_distance_list_it

```
int acp_distance_list_it(acp_list_it_t first, acp_list_it_t last);
```

Query for the distance between two iterators in a list.

Parameters:

   first   Iterator of a list

   last   Another iterator of the list

Return values:

   Distance between two iterators

4.5.31 acp_increment_list

```
acp_list_it_t acp_increment_list_it(acp_list_it_t it);
```

Increment iterator of a list.

Parameters:

   it   Iterator of a list

Return values:

   Next iterator of *it*

4.6 Map

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

4.6.1 Data types and macros

Data types:

```
typedef struct {
        acp_ga_t ga;
        uint64_t num_ranks;
        uint64_t num_slots;
} acp_map_t;

typedef struct {
        acp_map_t map;
        int rank;
        int slot;
        acp_ga_t elem;
} acp_map_it_t;

typedef struct {
        acp_map_it_t it;
        int success;
} acp_map_ib_t;
```

4.6.2　acp_create_map

`acp_map_t acp_create_map(int num_ranks, const int* ranks, int num_slots, int rank)`

Map creation.

Creates a map type data on any set of processes.

Parameters:

num_ranks	Number of processes.
ranks	Array of the rank numbers of the processes to distribute map.
num_slots	Number of slots.
rank	Rank number to place the information of the map.

Return values:

member ga == ACP_GA_NULL	Fail
otherwise	A reference of created map data.

4.6.3　acp_clear_map

`void acp_clear_map (acp_map_t map)`

Delete elements of lists in a map type data.

Parameters:

map　A reference of a map to clear.

4.6.4　acp_destroy_map

`void acp_destroy_map (acp_map_t map)`

Destroys a **map** type data.

Parameters:

map　A reference of a map to destroy.

4.6.5 acp_insert_map

```
acp_map_it_t acp_insert_map(acp_map_t map, const acp_ga_t key, size_t
size_key, const acp_ga_t value, size_t size_value)
```

Inserts a key-value pair to a *map.*

Parameters:

map   A reference of a map type data.

key   Global address of the key of the key-value pair.

size_key   Size of the key.

value   Global address of the value of the key-value pair.

size_value Size of the value.

Return values:

member success == 0   Fail or the same key-value pair is already in the map.

otherwise   The item inserted to the map.

4.6.6 acp_find_map

```
acp_map_ib_t acp_find_map(acp_map_t map, const acp_ga_t key, size_t
size_key)
```

Find a key-value pair according to a key in a *map*.

Parameters:

map   A reference of a map type data.

key Global   address of the key to search.

size_key   Size of the key.

Return values:

member elem == ACP_GA_NULL   Fail

otherwise   The item found in the map.

4.6.7 acp_end_map

`acp_map_ib_t acp_end_map(acp_map_t map)`

Query for the iterator just after the tail element of a *map*.

Parameters:

    map   A reference of a map type data.

Return values:

    member elem == ACP_GA_NULL Fail

    otherwise   The iterator just after the tail element of a map.

4.7 Malloc

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

4.7.1 acp_malloc

```
acp_ga_t acp_malloc (size_t, int)
```

4.7.2 acp_free

```
void acp_free (acp_ga_t)
```

4.8   Global Memory Management

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

4.8.1   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.

4.8.2   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.

4.8.3 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

4.8.4 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.

4.8.5 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

4.8.6 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

4.8.7   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

4.8.8   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.

4.8.9 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.

4.9　Global Memory Access

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

4.9.1　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.

4.9.2 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.

4.9.3   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.

4.9.4 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.

4.9.5 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.

4.9.6 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.

4.9.7 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.

4.9.8 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.

4.9.9 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.

4.9.10 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.

4.9.11 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.

4.9.12 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.

4.9.13 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.

4.9.14 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.

4.9.15 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.

4.9.16 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.

Appendix A   Running ACP with Multiple MPI programs

A.1 Overview

Current ACP library supports mechanisms for connecting Multiple MPI applications (Multi-MPI ACP). This is similar to MPI-Spawn except spawning dynamical processes.

A.2 Installation

A.2.1   MPI Library

Install MPI library. Check PATH and LD_LIBRARY_PATH shell environment variables. Currently, OpenMPI 1.6.3 and OpenMPI 1.10.2 are tested.

A.2.2   ACP

Sources of Multi-MPI ACP is included inside of the ACP library. To install this library, add "--with-mpi" option to the configure command ACP.

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

Original ACP libraries are also installed by this procedure. If your machine supports Infiniband (IB), this configure command detects and sets up as default IB library.

A.3 Compiling Programs

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

- Sources that use both MPI and ACP functions

    C: macpcc, macpgcc

    C++:   macpc++, macpcxx

    Fortran: macpfc (not provided yet)

- Sources used pure ACP functions

    C: acpcc, acpgcc

    C++:   acpc++, acpcxx

    Fortran: acpfc (not provided yet)

Example:

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

If your machine supports IB and you want to choose UDP environment, compile with "-ndev UDP" option.

A.4 Running Programs

Multi-MPI ACP library supports macprun program to launch Multi-MPI ACP appliations.

Example:

```
shell$ macprun -np 6 inputfile
```

with the following inputfile:

```
 ----------------------------------------------------------------
 # Number of runtime environments
 3
 # Runtime environments
 acprun -smemsize 10240            # options can be specified
 mpirun                            # options can be specified
 mpirun                            # options can be specified
 # Number of processes
 1
 2
 3
 # Commands
 ./hello_world_acp                 # options can be specified
 ./hello_world_macp                # options can be specified
 ./hello_world_macp                # options can be specified
 ----------------------------------------------------------------
```

This launches one hello_world_acp application (APP0) consists with one process by acprun, and two hello_world_macp applications (APP1 and APP2) with two and three processes on localhost by mpirun. These three applications with six processes are connected by ACP library as shown in the following figure:

```
proc ID    0       1        2       3       4       5
         +-------+---------------+-----------------------+
         | APP0  |     APP1      |         APP2          |
         +-------+-------+-------+-------+-------+-------+
Middle   |       |      MPI      |        MPI            |
Layer    +       +-------+-------+-------+-------+-------+
         |                      ACP                      |
         +-------+-------+-------+-------+-------+-------+
NODE     |                    localhost                  |
         +-----------------------------------------------+
```
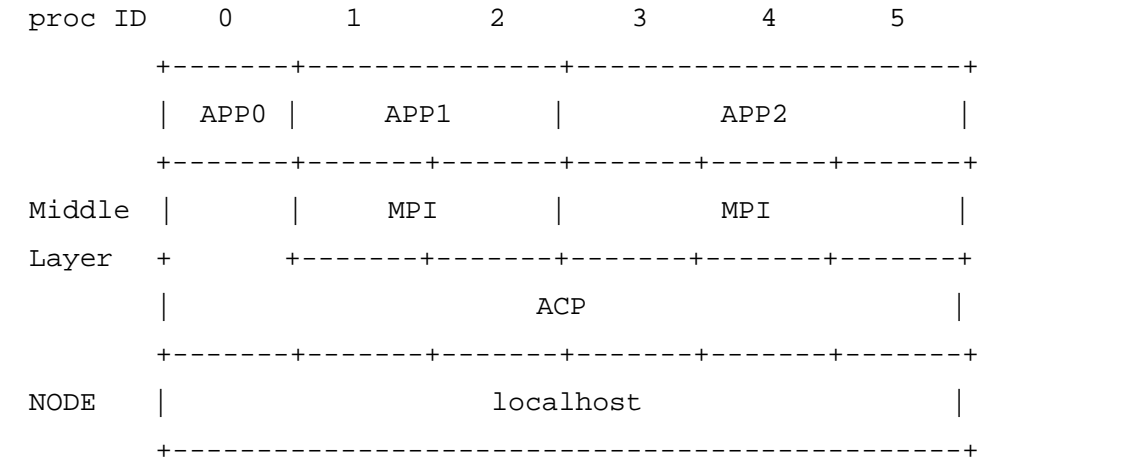
You can specify a "-nodefile nodefile" option to indicate hostnames on which hello_world_acp and hello_world_macp commands will be launched. If you want launch above processes on pc01 pc02 and pc03, use following command:

```
shell$ macprun -np 6 -nodefile nodefile inputfile
```

with the following nodefile:

```
----------------------------------------------------------------
pc01
pc01
pc02
pc02
pc03
pc03
----------------------------------------------------------------
```

This launches the programs as shown in the following figure:

```
proc ID     0       1       2       3       4       5

         +-------+--------------+----------------------+
         | APP0  |     APP1     |         APP2         |
         +-------+-------+-------+-------+-------+-------+
Midel    |       |     MPI      |         MPI          |
Layer    +       +-------+-------+-------+-------+-------+
         |                     ACP                     |
         +-------+-------+-------+-------+-------+-------+
NODE     |     pc01      |     pc02      |     pc03     |
         +---------------------------------------------+
```
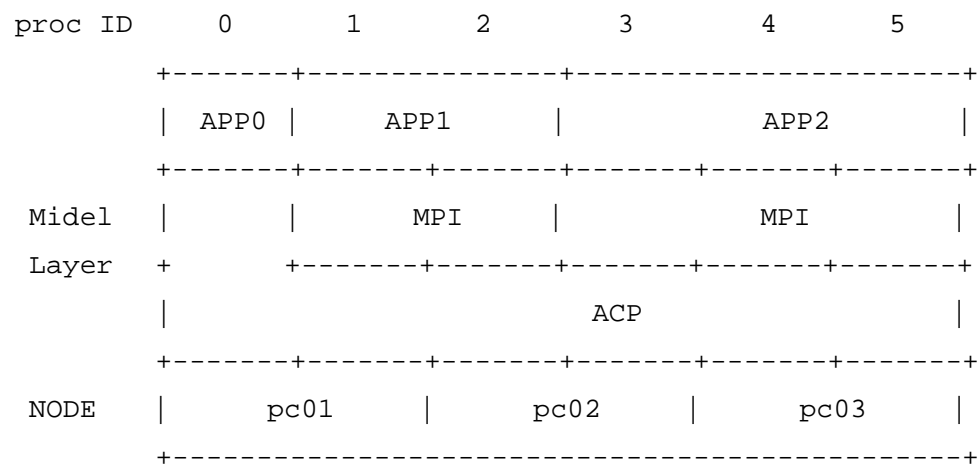
To specify network device, for example infiniband (or udp), use "-net ib" (or udp) option
as follows:

```
shell$ macprun -np 6 -nodefile nodefile -net ib inputfile
```