Advanced Communication Primitives (ACP) Library version 1.0

Tutorial

at SC14 (Nov., 2014)

Overview

- Motivation:
  Support applications to achieve both
  low memory consumption AND low overhead
  in communication.
  - Available memory per process is expected to be smaller.
  - Existing communication library:
    low memory OR low overhead.

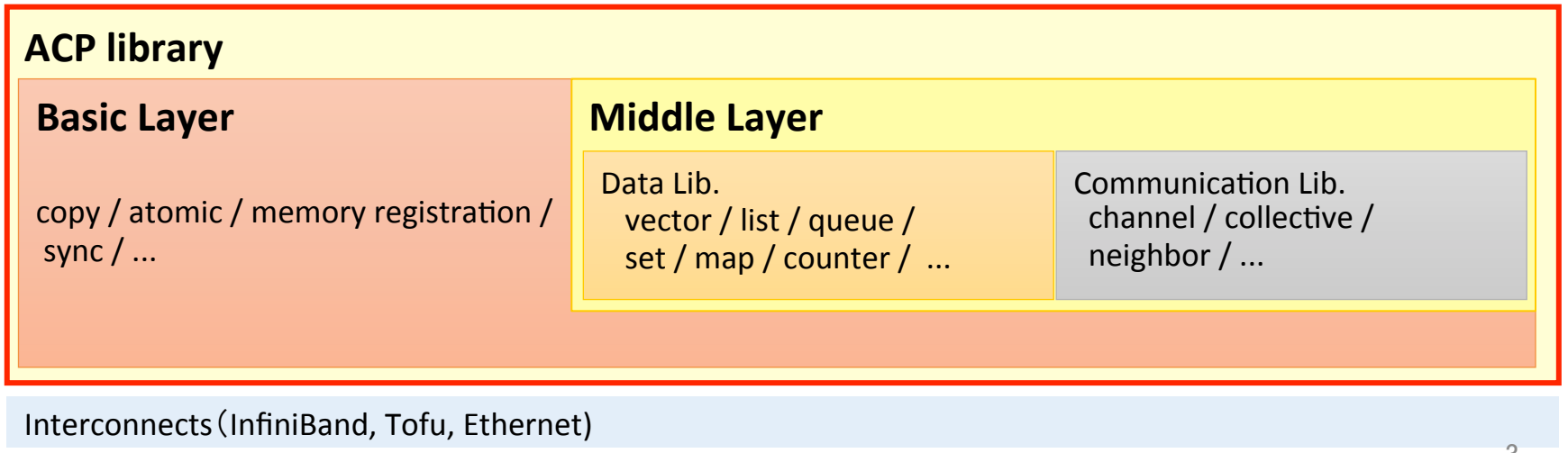
- **Advanced Communication Primitives** (**ACP**) Library:
  Designed to enable memory efficient communication.
  - Based on a thin layer that abstracts underlying interconnects.
  - Provide primitive interfaces as building blocks on the basic layer.
  - Each interface is independent and prepared on-demand.

Available at

http://ace-project.kyushu-u.ac.jp

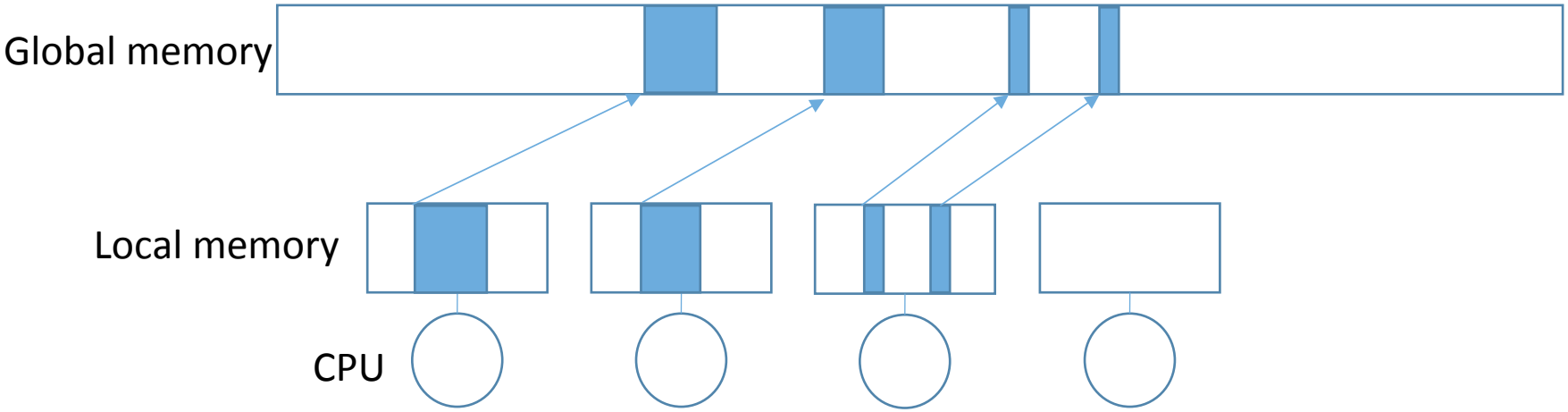
Structure

- Middle Layer
  - Interfaces for global data structures and communications.
- Basic Layer
  - Thin abstraction of underlying interconnects.
    - RDMA model.
  - Infrastructure interfaces.



Memory Model

- Local memory:
  - Ordinal address space of each process, managed by OS.
- Global memory:
  - Memory space virtually shared among processes.
    - Any local memory space of any process can be mapped to the global memory via registration.



Interfaces

- Infrastructure
  - initialization / finalization / synchronization / ranks
- Channel
  - message passing on Local Memory
- Data structure
  - list / vector / malloc on Global Memory
- Global Memory Management
  - registration / de-registration / query on Global Memory
- Global Memory Access
  - copy / atomic on Global Memory

A sample code of ACP with:
- infrastructure interface
- channel interface

```
#include <stdio.h>
#include <stdint.h>
#include "acp.h"

int main(int argc, char** argv)
{
  int rank, a;
  acp_ch_t ch;
  acp_request_t req;

  acp_init(&argc, &argv);
  rank = acp_rank();

  if (rank < 2) {
    ch = acp_create_ch(0, 1);
    if (rank == 0) {
      a = 100;
      req = acp_nbsend_ch(ch, &a, sizeof(int));
      acp_wait_ch(req);
    }
    if (rank == 1) {
      a = 0;
      req = acp_nbrecv_ch(ch, &a, sizeof(int));
      acp_wait_ch(req);
      printf("got %d\n", a);
    }
    req = acp_nbfree_ch(ch);
    acp_wait_ch(req);
  }

  acp_finalize();
  return 0;
}
```

Header

Channel handle

Request handle

Initialization

Query rank

Create channel

Start send

Wait

Start receive

Wait

Start Freeing channel

Wait

Finalize

Functions of Infrastructure Interface

- Initialization

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

- Finalization

```
int acp_finalize (void)
```

- Number of processes

```
int acp_procs (void)
```

- Process rank

```
int acp_rank (void)
```

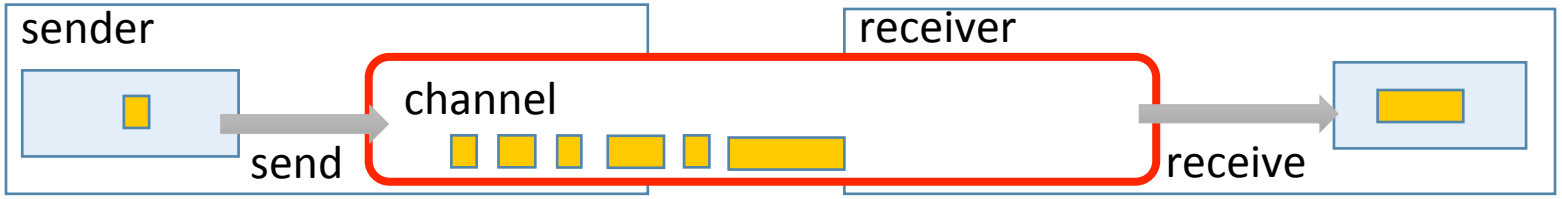
- Synchronization

```
int acp_sync (void)
```

Channel Interface

- Message passing via a 'channel'
  - Channel
    = a logical path constructed between sender and receiver.
  - Channels are explicitly allocated and de-allocated.

- Current available channel:
  single-directional, in-order channel
  - The sender and the receiver of a channel are fixed.
  - Guarantees the order of data transfers at both sides.



Functions of Channel Interface

- Channel creation

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

- Non-blocking channel free

```
acp_request_t acp_nbfree_ch (acp_ch_t ch)
```

- Non-blocking send

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

- Non-blocking receive

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

- Wait for completion of non-blocking operations

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

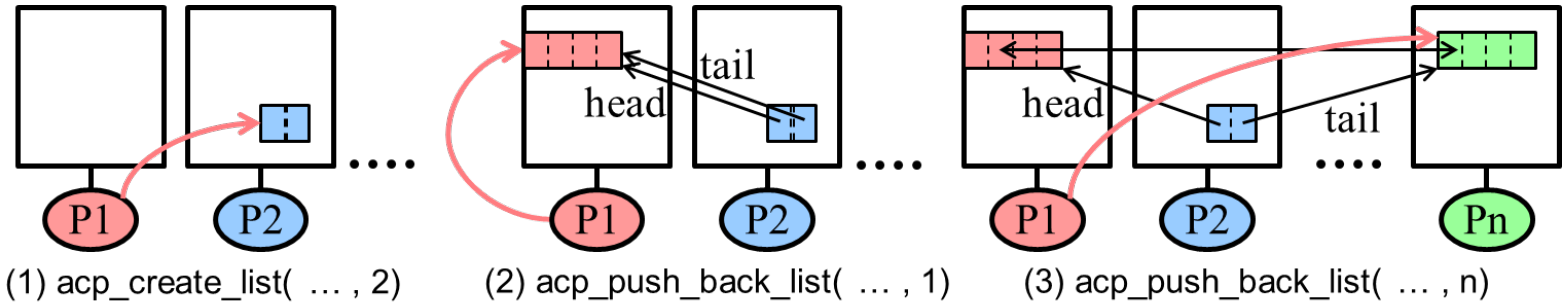
Some Details of Channels

- Channel creation:
  - Does not wait for the connection between sender and receiver.
    - Connection will be established by the first communication.
  - There can be redundant channels between the same sender and receiver.
- Channel free:
  - Once started to be freed, communications with that channel cause error.
- Receive:
  - If the size of the matching message is different from the specified size of receive, the smaller size is used.
- Wait:
  - Completion of non-blocking send ensures the send data safely stored away.

Data Structure Interfaces

- Support standard data structures on Global Memory
  - Vector, list and malloc are available, currently.
    - Planed structures: deque, map and set
  - Explicitly specifies the process of allocation.
    - Allocation, manipulation and de-allocation are asynchronous.

Sample of the manipulation of list:



Functions of Vector

- Create

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

- Destroy

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

- Duplicate

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

- Swap

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

Functions of List

- Create

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

- Destroy

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

- Insert

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

- Erase

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

- Add to tail

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

Functions of List (cont.)

- Query head

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

- Query tail

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

- Increment iterator

```
void acp_increment_list (acp_list_it_t * list)
```

- Decrement iterator

```
void acp_decrement_list (acp_list_it_t * list)
```

Functions of Malloc / Free

- Malloc

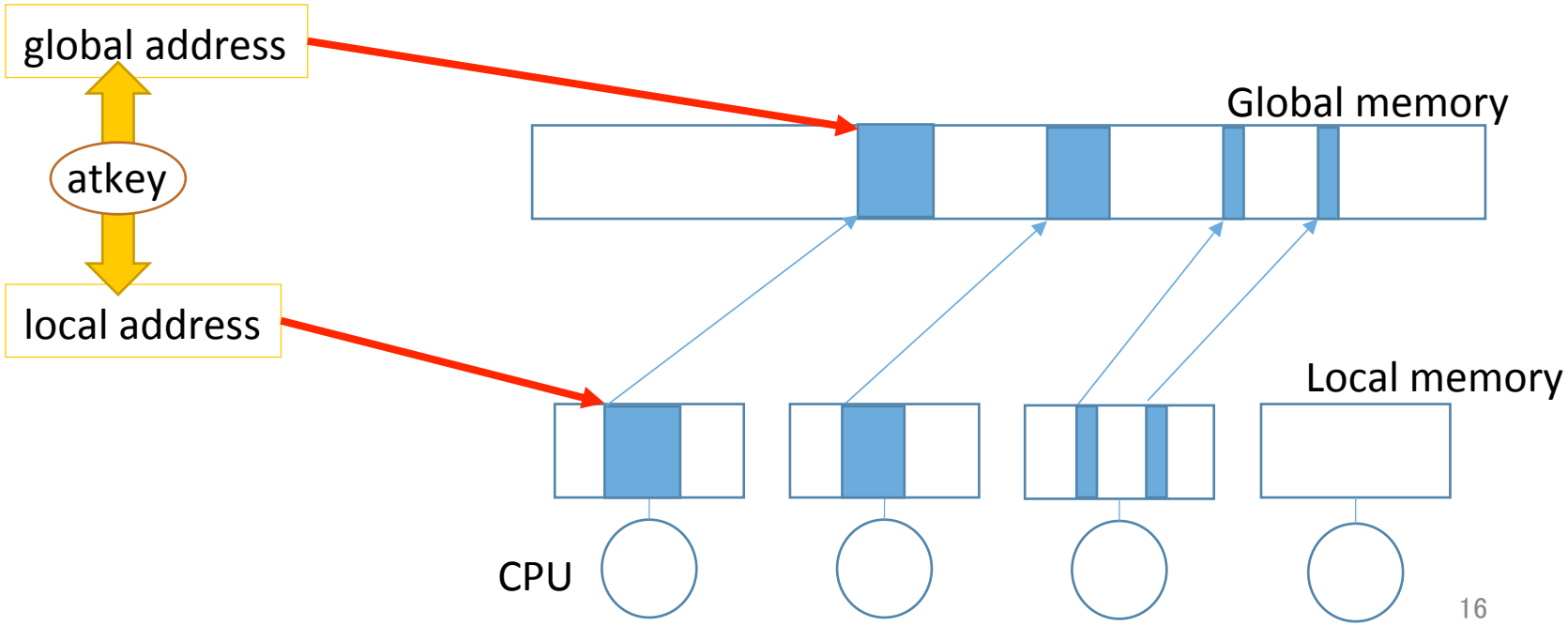
```
acp_ga_t acp_malloc (size_t size, int rank)
```

- Free

```
void acp_free (acp_ga_t)
```

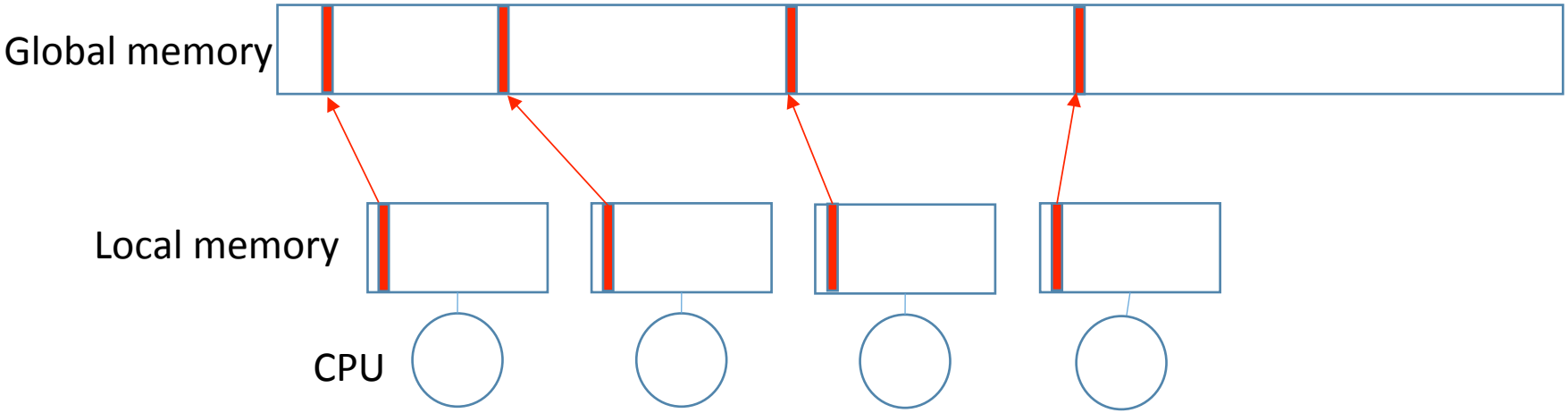
Global Memory Management

- Registration of Local Memory:
  - Creates an address-translation key (atkey).
- Global address:
  - Queried with the atkey and the local address.



Starter Memory

- A special memory region of each process that is registered at the initialization.
  - Global address can be queried directly.
  - Mainly used for exchanging global addresses.



Functions of Global Memory Management

- Register memory

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

- Unregister memory

```
int acp_unregister_memory (acp_atkey_t atkey)
```

- Query global address

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

- Query local address

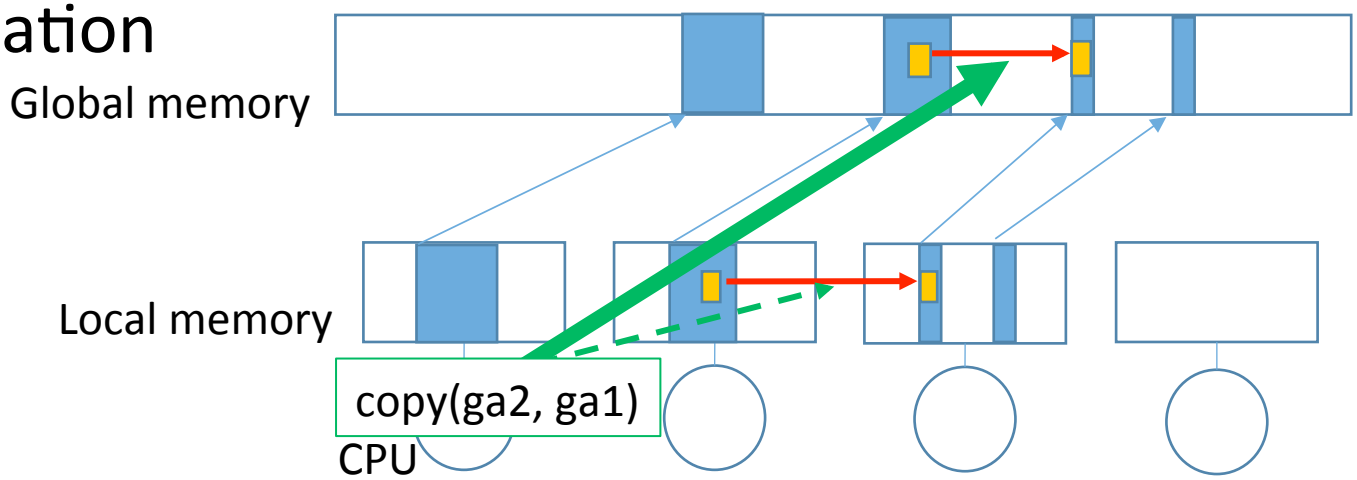
```
void* acp_query_address (acp_ga_t ga)
```

- Query global address of the starter memory

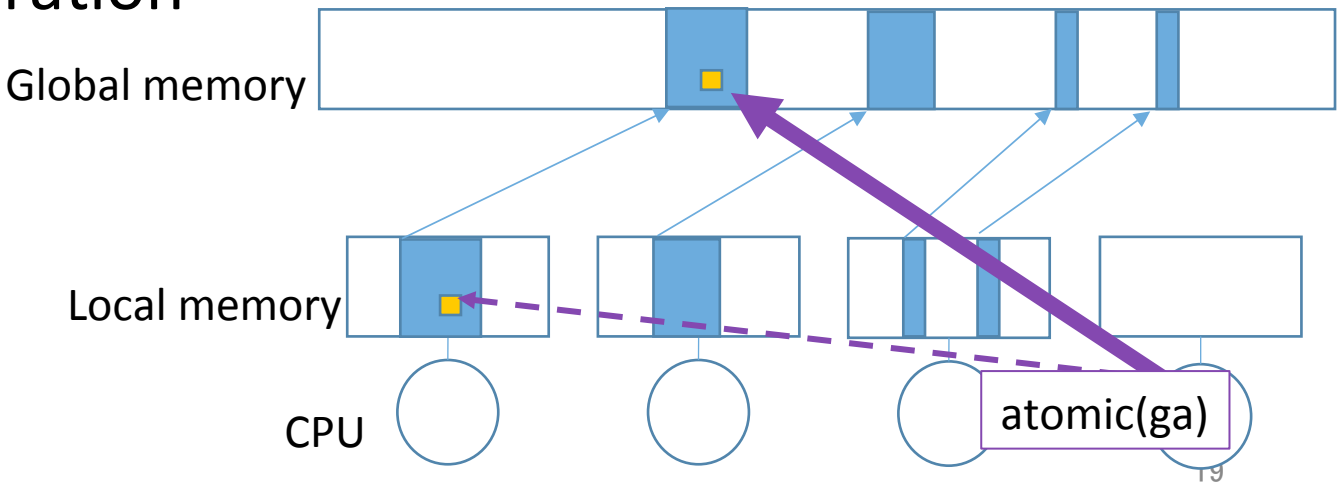
```
acp_ga_t acp_query_starter_ga (int rank)
```

Global Memory Access

- Copy operation



- Atomic operation



Functions of Global Memory Access

- Copy

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

- Atomic add (4byte)

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

- Atomic add (8byte)

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

- Other atomic operations:
  - CAS4, CAS8, AND4, AND8, OR4, OR8, XOR4, XOR8SWAP4, SWAP8
- Completion

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

- Inquery

```
int acp_inquire (acp_handle_t handle)
```

Some Details of Global Memory Access

- Non-blocking:
  - All global memory accesses are non-blocking.
  - Handle is used for waiting completion.

- 'Order' argument:
  - Specify the handle of the operation to be completed before starting this operation.
    - Used for describing algorithms of patterned communications.

- Completion and inquiry:
  - Completions are ensured in-order.
    - A completion of a handle ensures the completions of all the other global memory accesses invoked before that handle on the process.

Sample Code with Basic Layer

```
#include <stdio.h>
#include <stdint.h>
#include "acp.h"

int main(int argc, char *argv[])
{
  int myrank, procs, a, ret;
  acp_ga_t laga, raga, lstga, rstga, ragaga;
  acp_atkey_t akey, ragakey;
  acp_ga_t *lstla;
  acp_handle_t h;

  ret = acp_init(&argc, &argv);
  myrank = acp_rank();
  akey = acp_register_memory(&a, sizeof(int), 0);
  laga = acp_query_ga(akey, &a);

  if (myrank == 0) {
    a = 100;
    lstga = acp_query_starter_ga(0);
    lstla = acp_query_address(lstga);
    *lstla = laga;
    acp_sync();
  }
```

```
  if (myrank == 1) {
    rstga = acp_query_starter_ga(0);
    ragakey = acp_register_memory(&raga,
                                  sizeof(acp_ga_t), 0);
    ragaga = acp_query_ga(ragakey, &raga);
    acp_sync();
    h = acp_copy(ragaga, rstga, sizeof(acp_ga_t),
                                    ACP_HANDLE_NULL);
    acp_complete(h);
    h = acp_copy(laga, raga, sizeof(int),
                                    ACP_HANDLE_NULL);
    acp_complete(h);
    printf("got %d\n", a);
  }

  ret = acp_finalize();

  return 0;
}
```

Hands on

- Login
  - Use Putty to login to acecls.cc.kyushu-u.ac.jp
- Check source
  - $ less testch.c
  - $ less testbl.c
- Compile
  - $ acpcc testch.c -o testch
  - $ acpcc testbl.c -o testbl
- Submit
  - $ qsub test1.sh
  - $ qsub test2.sh
- Check result
  - $ less test1.sh.o?????
  - $ less test2.sh.o?????
- Modify
  - emacs testch.c
    or
  - emacs testbl.c

Questions?

- Contact:
  Takeshi Nanri (Kyushu Univ.)
  nanri@cc.kyushu-u.ac.jp