BoF 15:
How to achieve memory-efficient communication towards Exascale HPC

Takeshi Nanri (Kyushu Univ.)

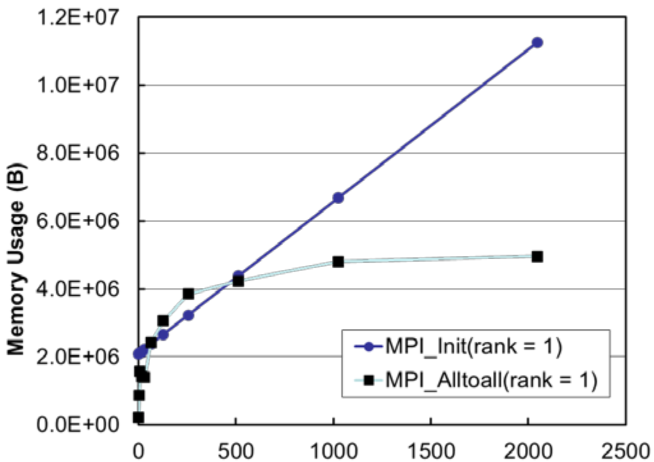
ISC2015
15 July, 2015
14:15 – 15:15

About our project
ACE (Advanced Communication lib. for Exa)

- A project for developing a scalable communication library with memory-efficient communications and runtime optimizations.
  - Supported by Japan Science and Technology Agency (JST)
- Duration: Oct 2011 - Mar 2017
- Members:
  - Kyushu Univ.
    - T. Nanri, T. Takami, H. Honda, R. Susukita, T. Kobayashi and Y. Morie
  - Fujitsu Ltd.
    - S. Sumimoto, Y. Ajima, K. Saga, T. Nose and N. Shida
  - ISIT Kyushu
    - H. Shibamura and T. Soga
  - Kyoto Univ.
    - K. Fukazawa

Memory Consumption in MPI Library

- Memory consumption in MPI_Init:
  Estimated 10GB/procs on 10M procs.
  - with default settings of Open MPI



MPI_Init vs. MPI_Alltoall

- Available memory per process:
  Predicted to be 10 to 100GB/procs.

- Efforts for reducing memory consumptions in MPI libraries:
  - Shared Receive Queues
  - Dynamic Connection
  etc.

by Paying additional overheads.

Our motivation:
Low memory and low overhead.

Two basic communication models

- PGAS
  - Put, Get, Atomic, Copy
  - **Asynchronous and Memory efficient**
  - **Require preparations**
    - Register memory, Exchange address
  - **Require codes for consistency**
    - Dependencies among producers and consumers
    - Mutual exclusion

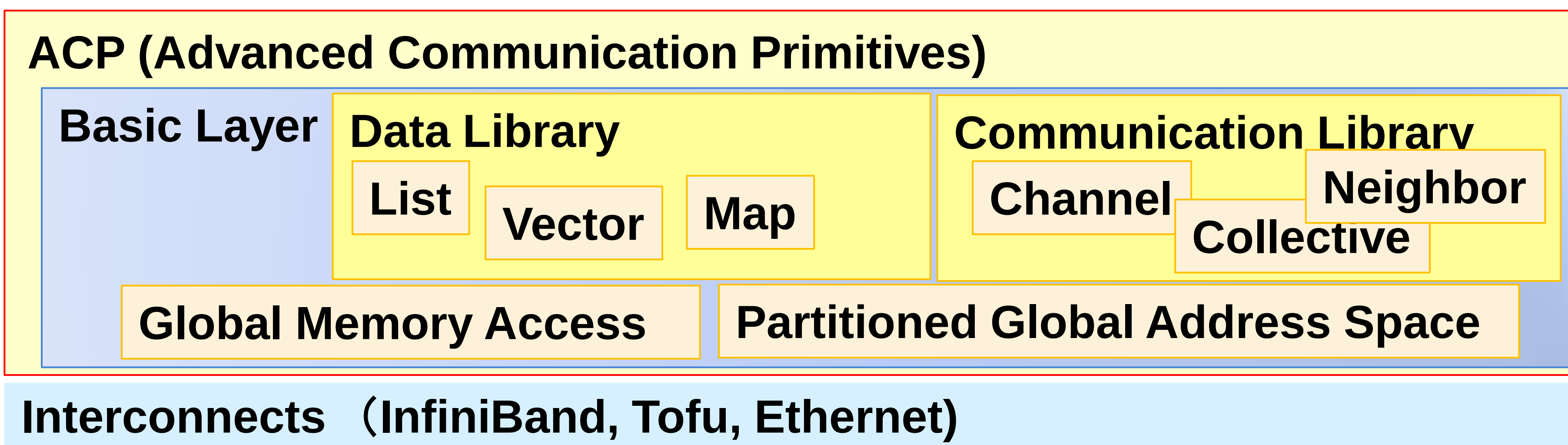
We want much higher programmability.

- Message Passing
  - Send/Receive, Collective, etc.
  - Synchronous
  - **Natural expression of producers and consumers**
  - **Memory consumption linear to the number of peers**

We still want this,
but in a memory-efficient way.

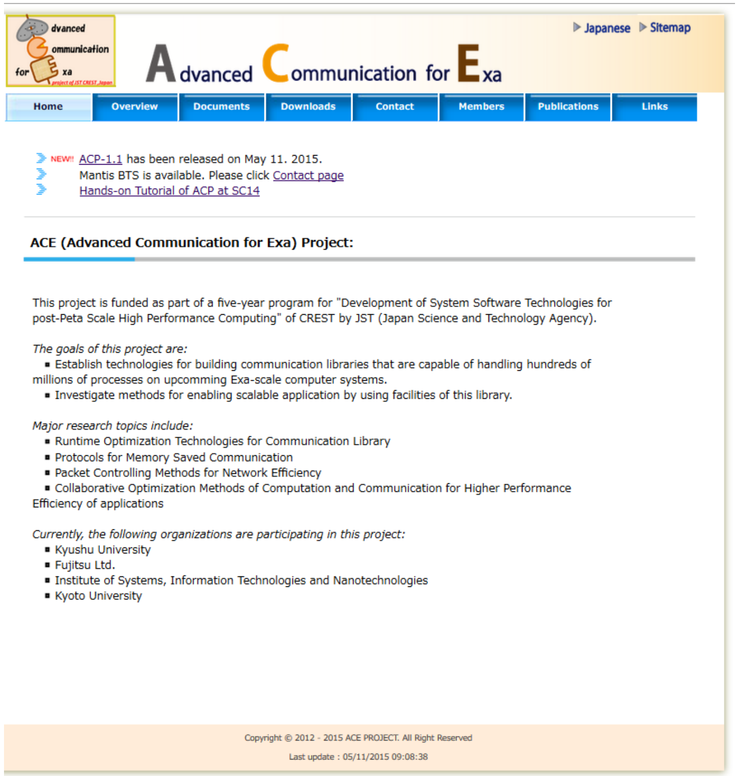
ACP (Advanced Communication Primitives) Library

- Collection of "primitive" communication operations.
- Basic Layer:
  GMA (Global Memory Access)-based thin abstraction of underlying interconnects for portability.
- Data Library:
  C++ STL-like data structures on PGAS.
- Communication Library:
  Message passing with user-controlled connections.



ACP Library is available

- Site:
  http://ace-project.kyushu-u.ac.jp
  - Search: ACE project Communication
- Contributions are welcome.
  - Applications
  - Proposal of new interfaces
    etc.



Topics in this BoF

- Shinji Sumimoto
  "PGAS model in a low level communication layer"
- Yuichiro Ajima
  "Distributed data structure interface"
- Yoshiyuki Morie
  "Explicit user-controlled connection"
- Toshiya Takami
  "Application development using the ACP library"

BoF 15-2:
PGAS Model in a Low Level Communication Layer

Shinji Sumimoto
(Fujitsu/JST-CREST)

ISC2015

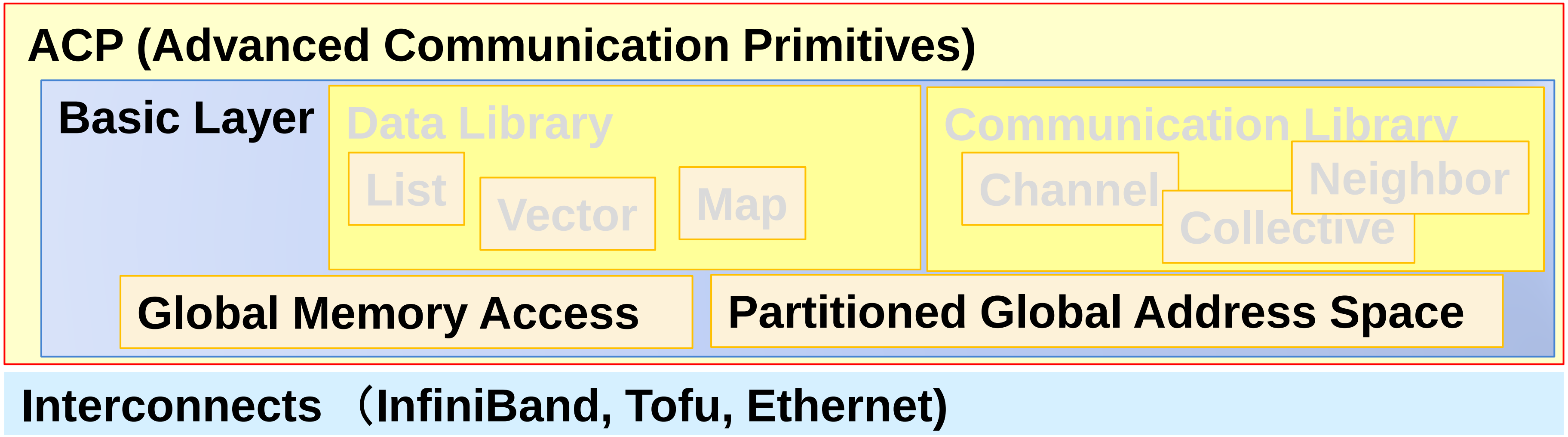
15 July, 2015

Outline of This Talk

- Design Policies of ACP Basic Layer
- ACP Global Memory Access(GMA) Design
- ACP Basic Layer(ACPbl) Overview

Design Policies of ACP Basic Layer

- Design Policies:
  - For Memory Efficient Communication：　Explicitly Controlled
    - Implicit Control is difficult to realize least memory usage
  - For Low Latency Data Exchange：　RDMA Based
    - Low Latency and Data Exchange without CPU processing
- ACP Basic Layer Provides RDMA based Global Memory Access



Goal and Requirements of GMA

- Goal: Accessing Distributed Global Memory like a Local Shared Memory
  - No need to recognize location of memory
- Requirements of GMA:
  - GMA enables to build distributed and complicated data structure on PGAS
  - Asynchronous data processing on PGAS
  - Memory usage must be explicitly controlled by Users

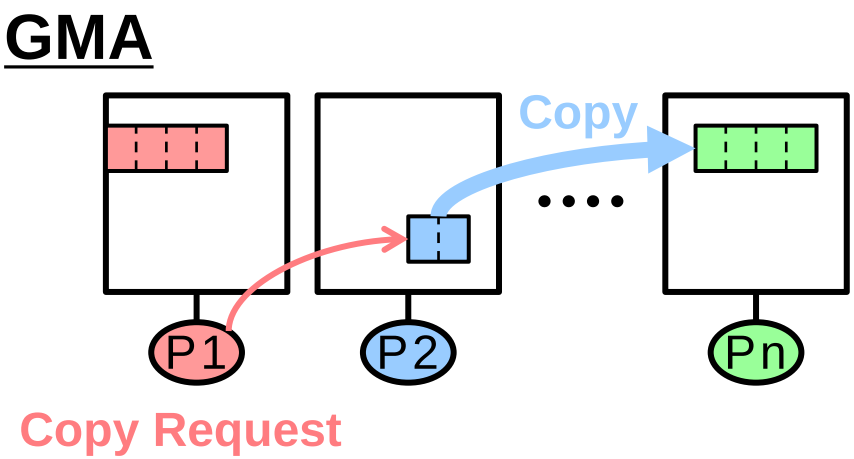
Solutions for the GMA Requirements

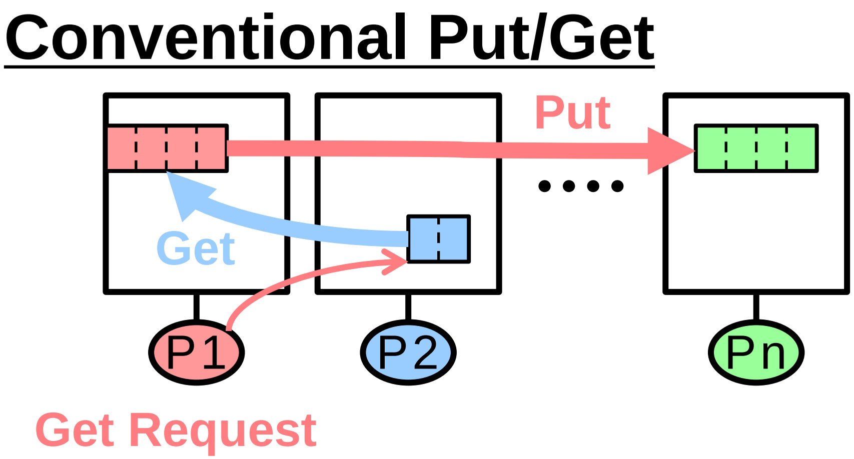
- Using 64 bit Global Memory Address so that remote atomic memory operation can be used.
  - Address Formats are different among networks because they need to have endpoint and real memory address information.

- Providing Memory Registration APIs
  - Users allocate local memory and register it to Global Memory.

ACP Basic Layer: Data Transfer Method

- Global Memory Access: GMA
  - Global Memory Copy between Distributed Processes including Non-Local Processes
  - Aimed to Optimize Asynchronous Data Transfer with Complicated Communication Pattern
- Existing Work: MPI Remote Memory Access(RMA)
  - Data Transfer between Distributed Memory and Local Memory Controlled by source or destination process.





ACP Basic Layer APIs

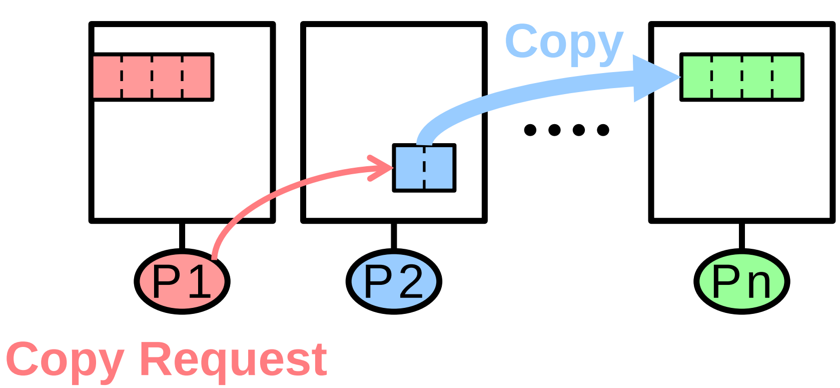
- Supporting Infrastructure, Global Memory Management and Access Functions.

	ACP Basic Layer APIs
Infrastructure	**acp_init, acp_finalize, acp_reset, acp_abort, acp_sync, acp_rank, acp_procs**
Global Memory Management	**acp_register_memory, acp_unregister_memory, acp_query_ga, acp_query_starter_ga, acp_query_address, acp_query_rank, acp_query_color, acp_colors**
Global Memory Access	**acp_copy, acp_complete, acp_inquire, acp_cas4, acp_cas8, acp_swap4, acp_swap8 etc..**

Data Transfer API: acp_copy

```
acp_handle_t acp_copy(acp_ga_t dst, acp_ga_t src,
                      acp_size_t size, acp_handle_t order);

Argument:
   dst: Destination Global Address
   src: Source Global Address
   size: Copy Size
   order: Specify acp_handle_t or ACP_HANDLE_ALL/NULL
Return Value:
   != ACP_HANDLE_NULL: GMA Handle
   == ACP_HANDLE_NULL: Fail
```

```
acp_handle_t: ACP Handle
acp_ga_t:     ACP Global Address
acp_size_t:   ACP Data Size
```



ACP Sample Code: Asynchronous Allgather

- Asynchronous n byte Allgather Data Transfer
  - Each process broadcasts data to the other processes by using binary tree algorithm
  - The broadcast of each process is processed asynchronously.

```
rank = acp_rank();
procs = acp_procs();
handle[rank] = ACE_HANDLE_NULL;
for ( i = 1 ; i < procs ; i++ ) {
    dst = (rank +  i    ) % procs;
    src = (rank + (i>>1)) % procs;
    handle[dst] = acp_copy(
        acp_query_startar_ga(dst) + n * rank,
        acp_query_startar_ga(src) + n * rank,
        n,
        handle[src]
    );
}
acp_complete(ACP_HANDLE_ALL);
acp_sync();
```

ACP Basic Layer(ACPbl) Implementations

- Initial implementations of the ACPbl:

  UDP (Ethernet), InfiniBand (IB) and Tofu Interconnect Versions

  - UDP Version: Developed as a Reference Implementation for validation
  - IB and Tofu Versions: Developed as Practical Execution Environments

  All implementations create a communication thread for each process to emulate copy and atomic operation

ACPbl: Memory Usage Estimation@ 1M Procs.

	Tofu	UDP
Per Number of Processes	69MB@1M Processes • Command Receive Buffer 64B • Tofu Address Table 4B • Tofu Routing Table 1B	18MB@1M Processes • IP Address 4B、Port Number 2B • Send Sequence Number 4B、Receive Sequence Number 4B • Process Number 4B
Per Memory Entry	9KB@128 Entries • Registration Table 40B • Address Lookup Table 16B/Proc. ＋ 8bytes×256（Fixed）	
Misc	262KB • Command Queue and Buffers 64B×4,096	647KB • Command Queue 80B×4,096 • Command Station 1,504B×64 • Delegate Station 3,480×64
Total@1M Processes	約70MB	約19MB

Only 70MB of memory for 1 Million Processes on Tofu

ACP Evaluation Environment

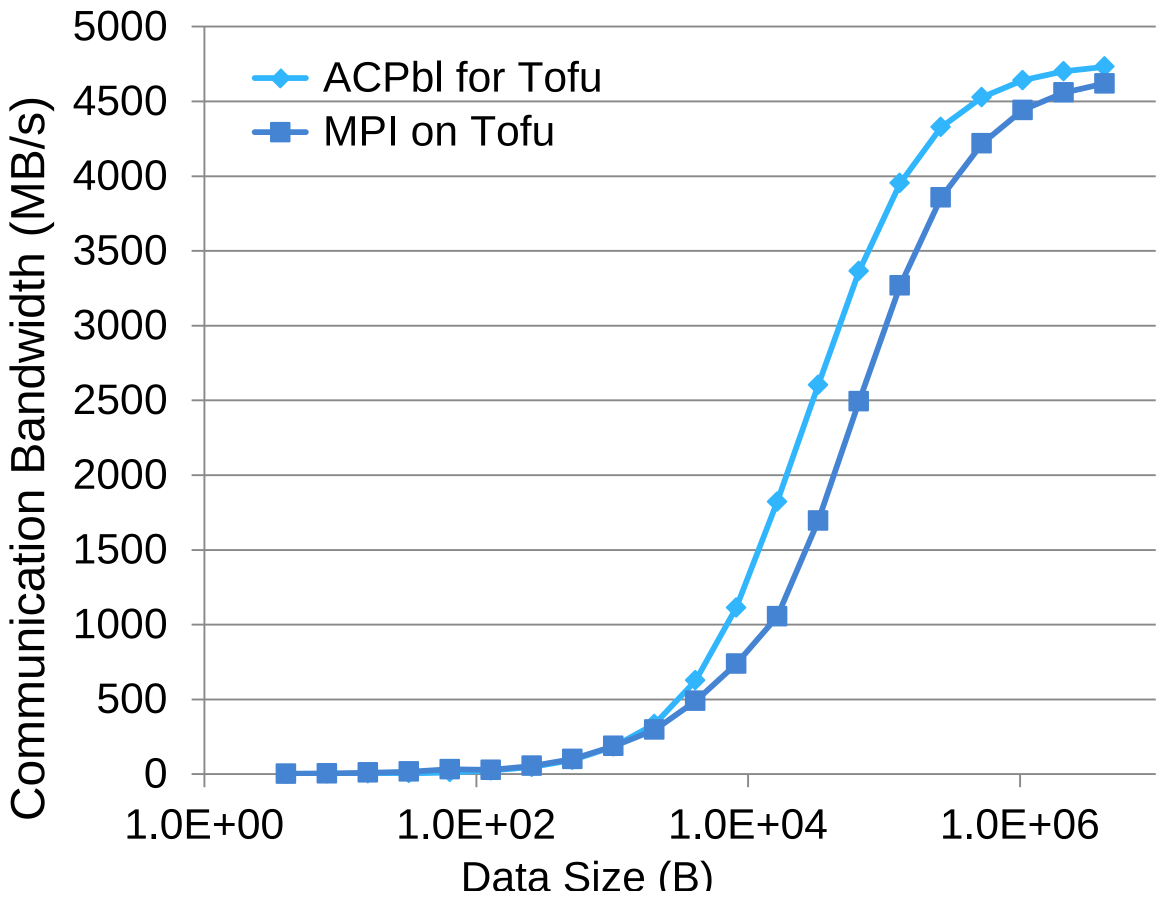
- Preliminary Evaluations
  - ACPbl(UDP): PC cluster and Gigabit Ethernet (GbE)
  - ACPbl(Tofu): PRIMEHPC FX10 with Tofu

- Three Node Evaluation for performing remote-to-remote copy

	PC Cluster	Supercomputer
Node	Fujitsu PRIMERGY RX200 S5	Fujitsu PRIMEHPC FX10
CPU	Intel Xeon E5520, 2 sockets (4 cores, 2.27 GHz)	Fujitsu SPARC64TM IXfx (16 cores, 1.848 GHz)
Memory	48GB, DDR3 1066MHz	64GB, DDR3 1333MHz
Network	Gigabit Ethernet (125 MB/s)	Tofu interconnect (5.0 GB/s)
OS	Linux version 2.6.32	Linux version 2.6.52.8

Preliminary ACP Basic Layer Performance

- Reference Implementation has been finished and ACP Basic Layer performance is better than that of MPI.



ACP Basic Layer Performance on Tofu

Evaluation System:
System: Fujitsu PRIMEHPC FX10
CPU: SPARC64^{TM} IXfx
Memory: 32GB/node

Summary

- ACP Basic Layer:
  - For Memory Efficient Communication：Explicitly Controlled
  - For Low Latency Data Exchange：RDMA Based

- How to achieve memory-efficient communication towards Exascale HPC
  - It should be PGAS Based
  - ACP provides PGAS based low level APIs as native communication primitives

shaping tomorrow with you

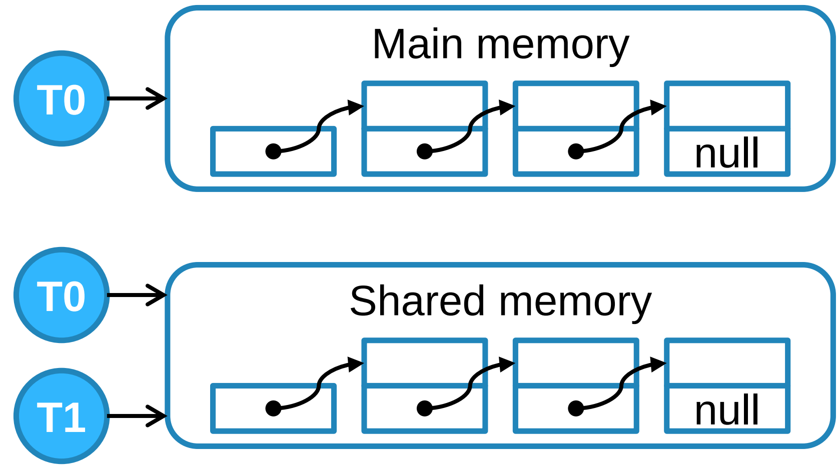
Distributed Data Structure Interfaces

Yuichiro Ajima (Fujitsu)

ISC2015
15 July, 2015

Data Structure

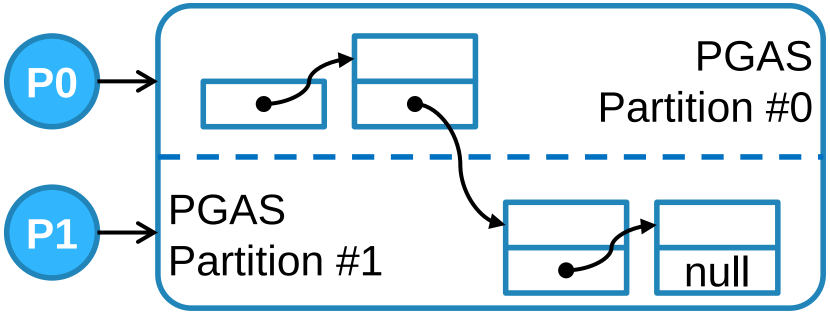
- Ordinary data structure
  - Main memory
  - Single thread
- Concurrent data structure
  - Shared memory
  - Multiple threads



- Why data structure?
  - To handle irregular data
    - Ex. particle methods, sparse matrices, unstructured meshes
  - For sophisticated memory management
    - Reduce, Reuse, Recycle

Distributed Data Structure on PGAS

- Distributed data structure
  - Distributed memory
  - Multiple processes
  - Partitioned Global Address Space



- Why PGAS?
  - Innate locality-awareness for performance
  - Global address is easy to implement reference semantics
    - Allowing wrapper to be written for high-level language

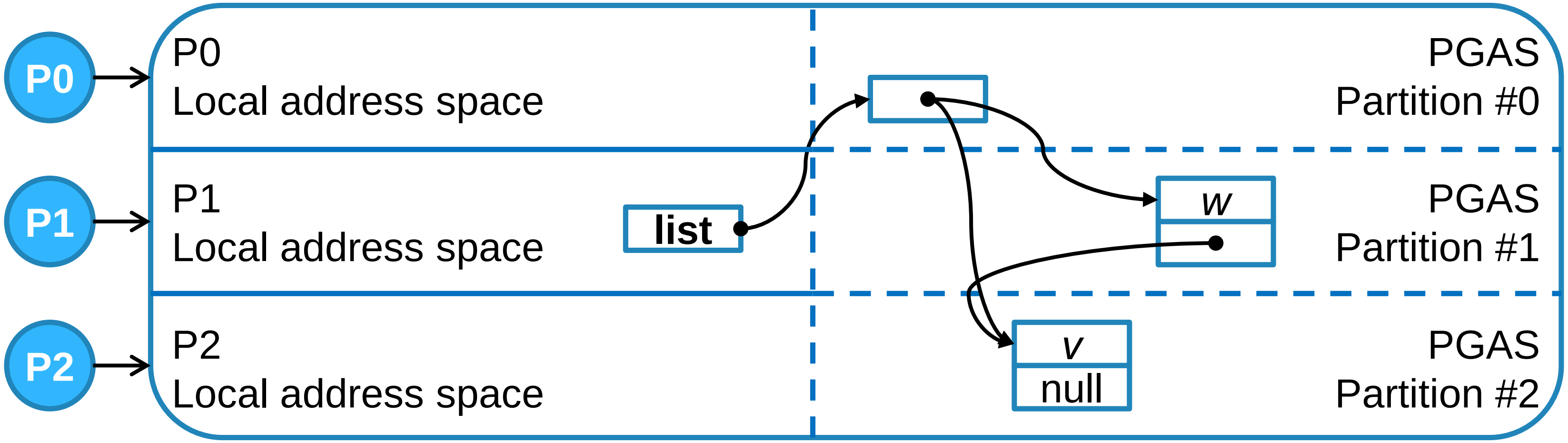
Data Library of ACP

- Similar to C++ Standard Template Library
  - But data-type agnostic C library
- Five data structure types
  - acp_vector_t variable-length array
  - acp_list_t bi-directional linked-list
  - acp_deque_t bi-directional queue
  - acp_set_t unordered collection
  - acp_map_t unordered key-value store
- Iterators
  - Ex. acp_vector_it_t, acp_list_it_t, acp_map_it_t
- Functions
  - Ex. acp_create_vector(), acp_destroy_list(), acp_insert_map()

Data Placement Control

- Each object or element can be placed on different process ranks
- Example) executed at P1

```
list = acp_create_list(0); // on #0
acp_push_back_list(list, &v, sizeof(int), 2); // on #2
acp_push_front_list(list, &w, sizeof(int), 1); // on #1
```



Iterator

- An iterator is a reference to an element of an object
  - Iterator abstracts index and pointer to an element
  - Iterator also contains the object information
- Function **begin** provides the iterator of the first element
- Function **end** provides the iterator of the end sentinel
- An iterator can be **increment**ed, **decrement**ed or **dereferenc**ed

Ex. Iterator functions

---

acp_<type>_it_t **acp_begin_<type>**(acp_<type>_t *object*);

acp_<type>_it_t **acp_end_<type>**(acp_<type>_t *object*);

acp_<type>_it_t **acp_increment_<type>**(acp_<type>_it_t *iter*);

acp_ga_t **acp_dereference_<type>**(acp_<type>_it_t *iter*);

Iterator – Sample Code

- To count number of keys in a set

```
acp_set_t set;
acp_set_it_t iter;
int count;
…

count = 0;
for (iter = acp_begin_set(set);
     iter != acp_end_set(set);
     iter = acp_increment_set(iter))
{
    count++;
}
```

Global Memory Allocator

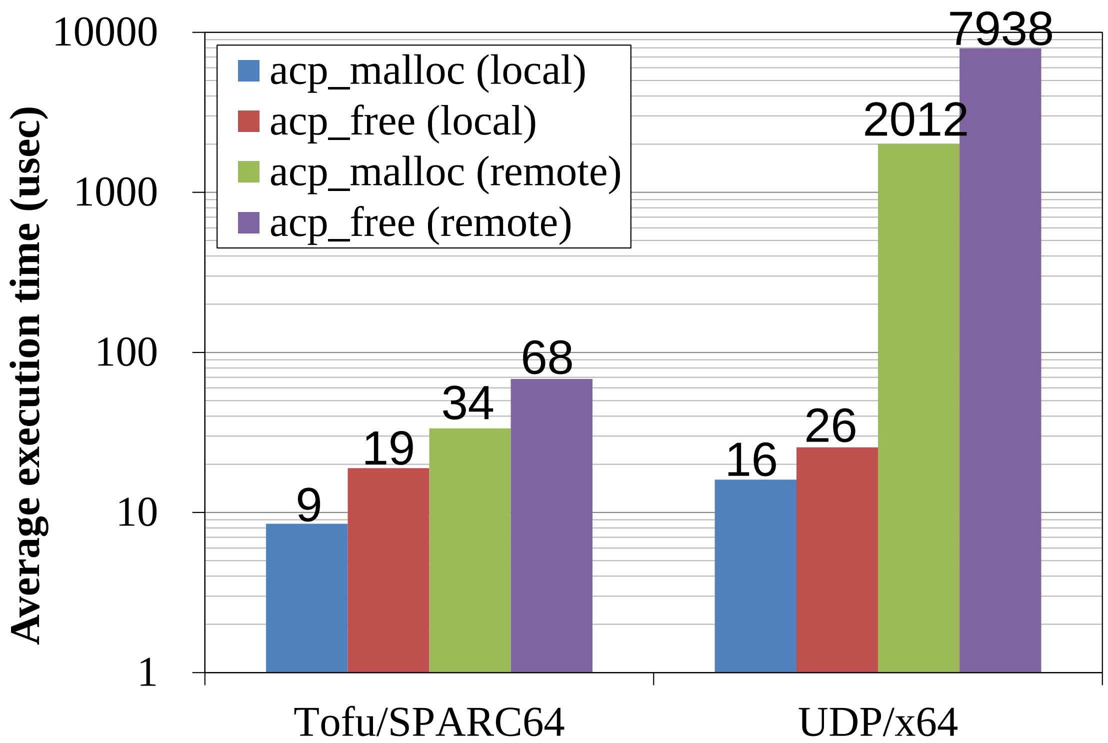
- Implemented for internal use
  - The Basic Layer only registers local memory
  - The Data Library requires dynamic allocator of remote memory
- acp_malloc()
  - Allocates specified size of remote memory on specified rank
  - Returns start global address of the allocated memory
- acp_free()
  - Deallocates an allocated memory specified by start global address

Global memory allocator functions

acp_ga_t **acp_malloc**(size_t *size*, int *rank*);

void **acp_free**(acp_ga_t *ga*);

Random Allocation/Deallocation Time



	Tofu/SPARC64	**UDP/x86**
CPU	Fujitsu SPARC64™ IXfx	Intel Xeon E5520
Network	Tofu interconnect (5.0 GB/s)	Gigabit Ethernet (125 MB/s)
ACP Basic Layer	Tofu version	UDP version

Development Status

- Completed
  - Global memory allocator: initial version
    - K&R malloc working on a dedicated static data segment
  - Data structure interfaces: partial implementation for evaluation
- On-going
  - Implementation of data structure interfaces
    - The initial version will be released in this year
- In discussion
  - Advanced memory allocator algorithms
  - Non-blocking data structure algorithms

Summary

- Data structure library
  - Sophisticated memory management algorithms for irregular data
- Distributed data structure on PGAS
  - Locality-aware and easy to implement reference semantics
- Data Library of ACP
  - Data-type agnostic C library
  - Data structure types are similar to those of C++ STL
- Data placement control
  - Each object or element can be placed on different process ranks
  - Iterator – reference to an element of an object
- Global memory allocator for internal use
- The initial version will be released in this year

EXPLICIT USER-CONTROLLED CONNECTION

Yoshiyuki Morie (Kyushu Univ.)

15 July, 2015

BoF: How to Achieve Memory-Efficent Communication towards Exascale HPC

ISC2015

Importance of message passing

- Message passing is easy to write

p0

x = *calc*
send **x** to p1

p1

receive **x** from p0 to **y**
calc(**y**)

- RDMA is not

p0

register **x**
notify remote address of **x**

x = *calc*
notify that **x** *is available*

p1

register **y**
achieve remote address of **x**

wait for the availability of **x**

get **x** *to* **y**
calc(**y**)

Connection of message-passing

- Specification allows message-passing at anytime with anyone

This is convenient, but…

- Once a connection is allocated, remains until the end
- Causes severe memory consumption for large number of processes

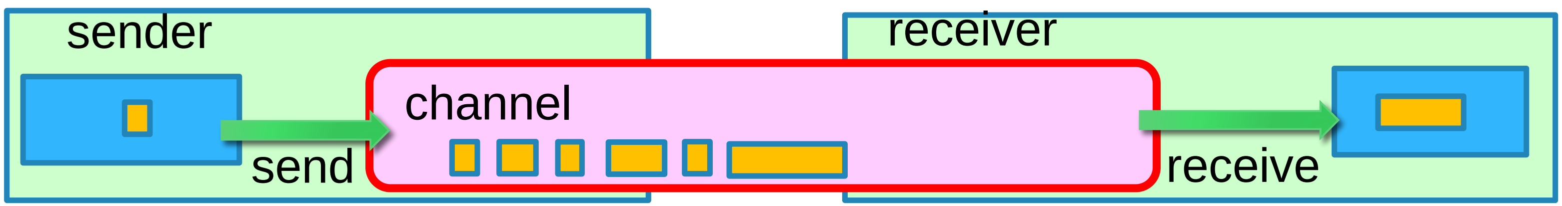
No explicit notification of 'no more communication with it'

solution

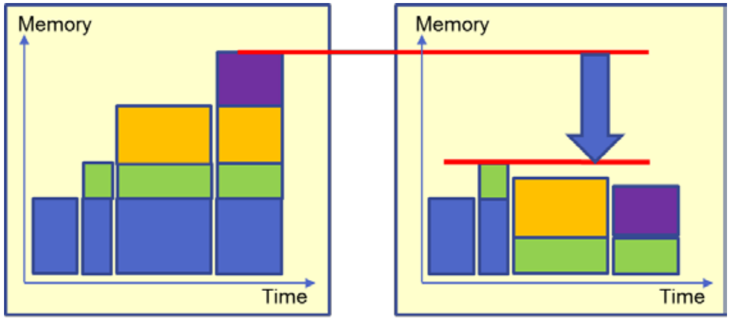
User controlled connection

User Controlled Connection in ACP Lib: ex) Channel Interface

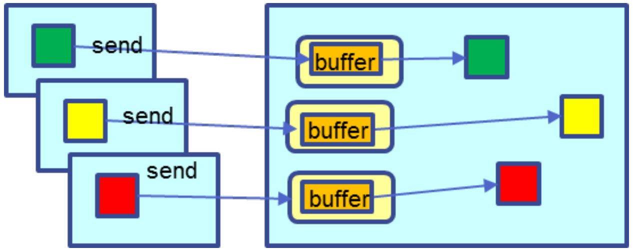
- A primitive interface for message passing
  - Single direction



- Low memory consumption
  - Allocate/Free



- and, Low overhead
  - Prepared



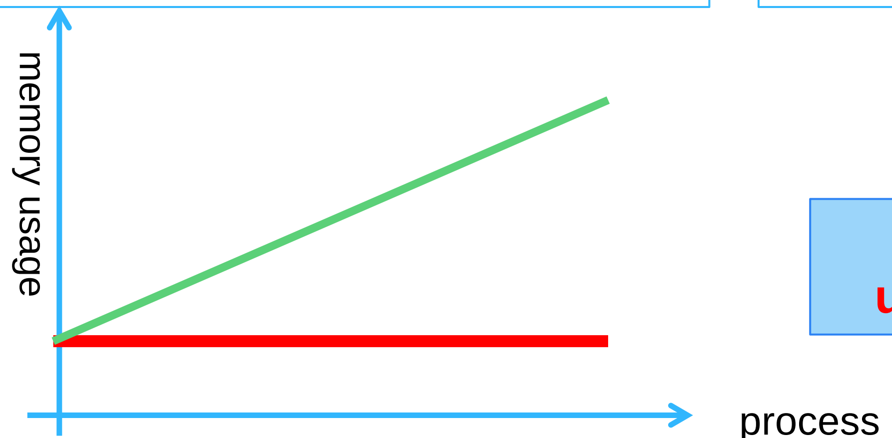
Functions of Channel Interface

- channel creation: **`acp_create_ch(src, dst)`**
  - prepares data structure, and start connection
- send/recv:
  `acp_nbsend_ch(ch,buf,size)`,
  `acp_nbrecv_ch(ch,buf,size)`
  - non-blocking message passing
- free channel: **`acp_free_ch(ch)`**
  - starts negotiation for freeing a channel
- wait for non-blocking operations: **`acp_wait_ch(req)`**

Sample program: master-worker

```
void master()
{
  loop{
    worker = pick_worker()
    ch = acp_create_ch(me, worker)

    data = create_data()
    req = acp_nbsend_ch(ch, data)
    acp_wait_ch(req)

    acp_free_ch(ch)
  }
}
```

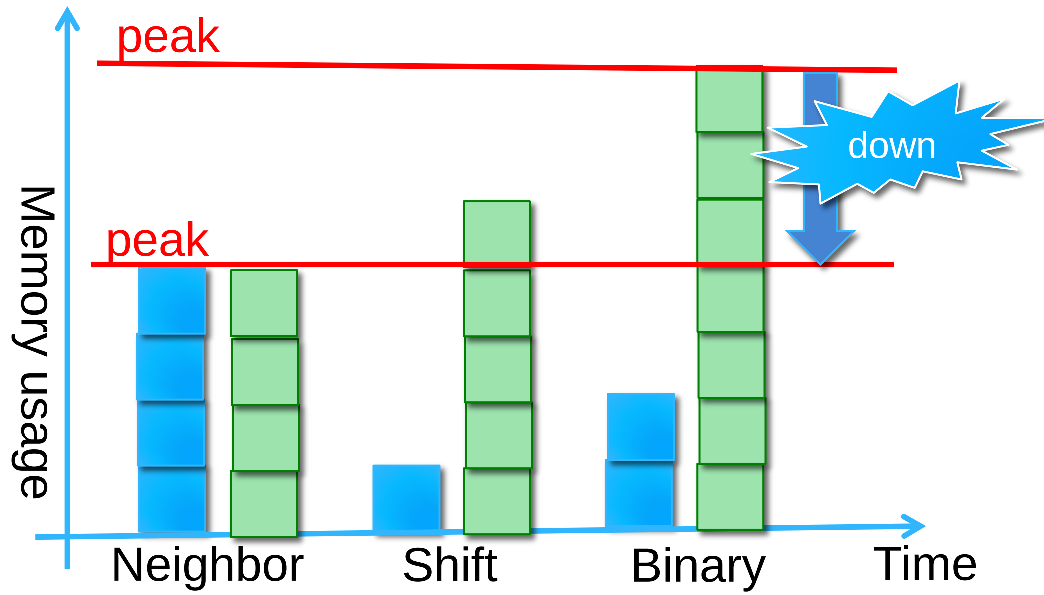
```
void worker()
{
  loop{
    ch = acp_create_ch(master,me)

    req = acp_nbrecv_ch(ch, data)
    calc2(data)
    acp_wait_ch(req)
    calc1(data)

    acp_free_ch(ch)
  }
}
```



It is important to release unnecessary connections

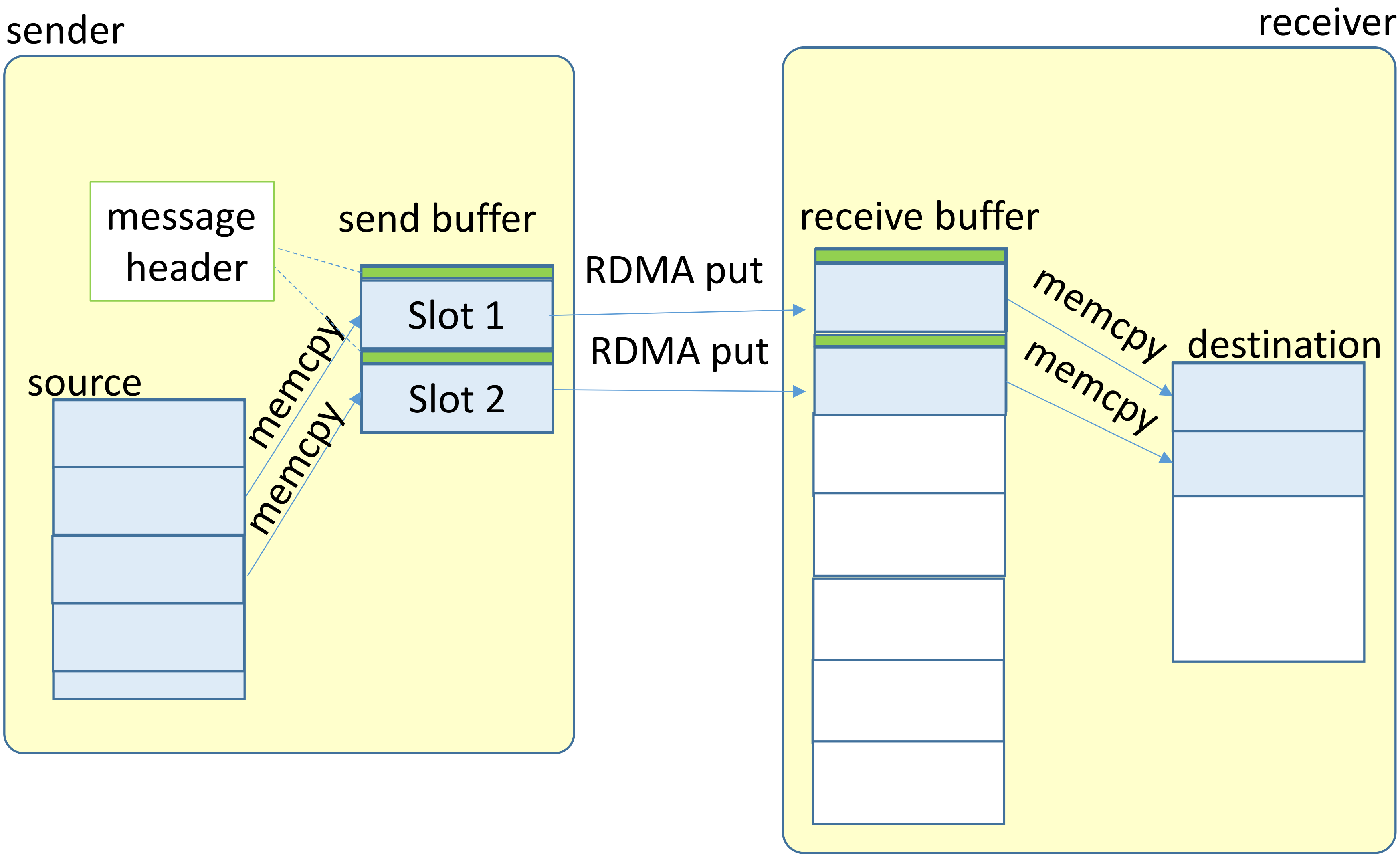
Sample program:
Halo (Neighbor) -> Shift -> Tree(Binary)

Connect Loop Start_comm Calc Wait_comm End loop Disconnection	Halo (neighbor)
Connect Loop Start_comm Calc Wait_comm End loop Disconnection	Shift
Connect Loop Start_comm Calc Wait_comm End loop Disconnection	Tree (binary)

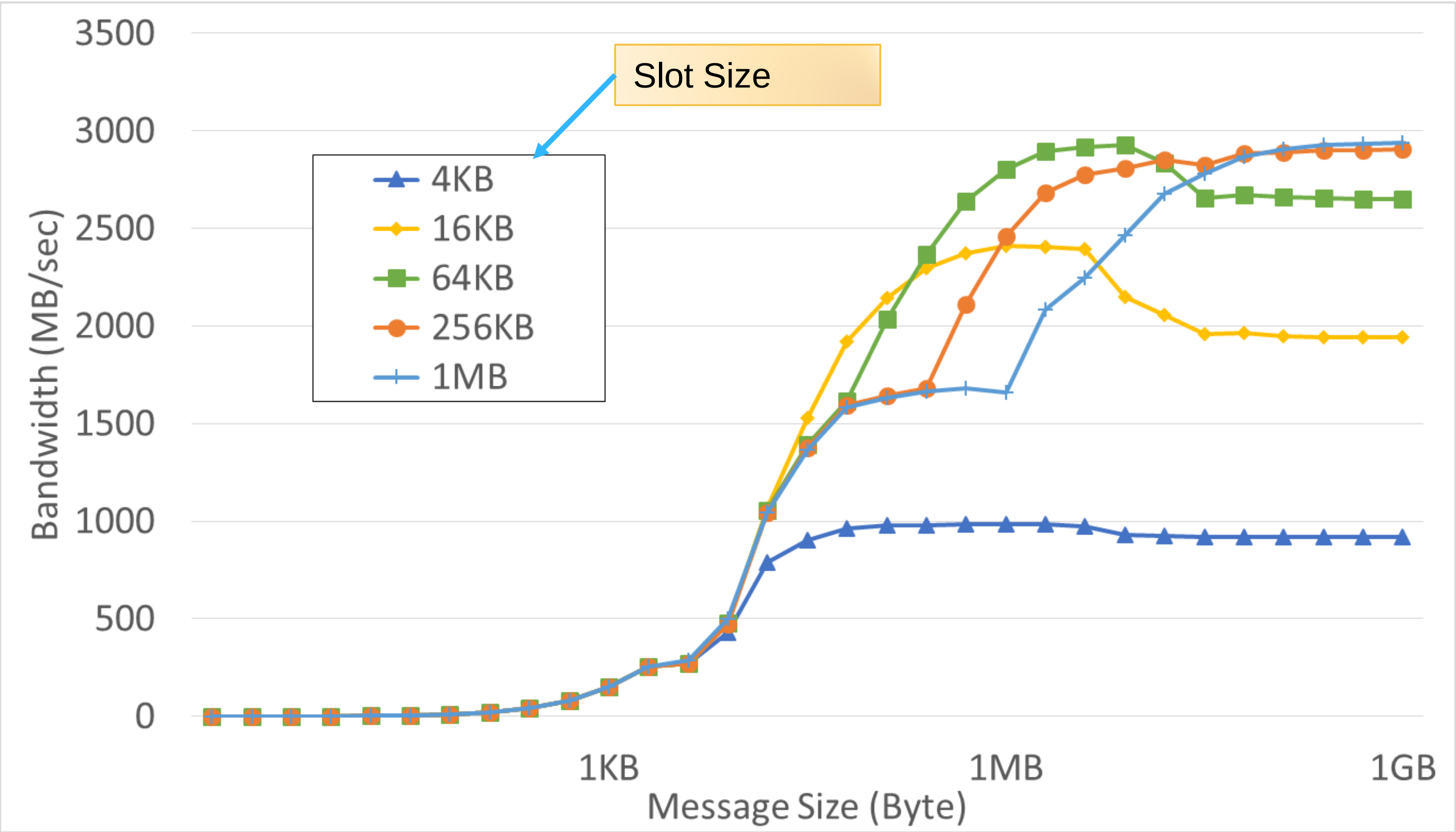


Minimize peak memory consumption

Implementation of Channel interface



Bandwidth



Environment:
PC Cluster (Xeon E5-2609, 8GB RAM, InfiniBand QDR)

Memory usage

Slot size	# of slots		
	2	4	8
4KB	8392	16616	33064
16KB	32968	65768	131368
64KB	131272	262376	524584
256KB	524488	1048808	2097448
1MB	2097352	4194536	8388904

- According to memory usage vs communication performance, channel interface should been implemented 64KB and 2 slots.

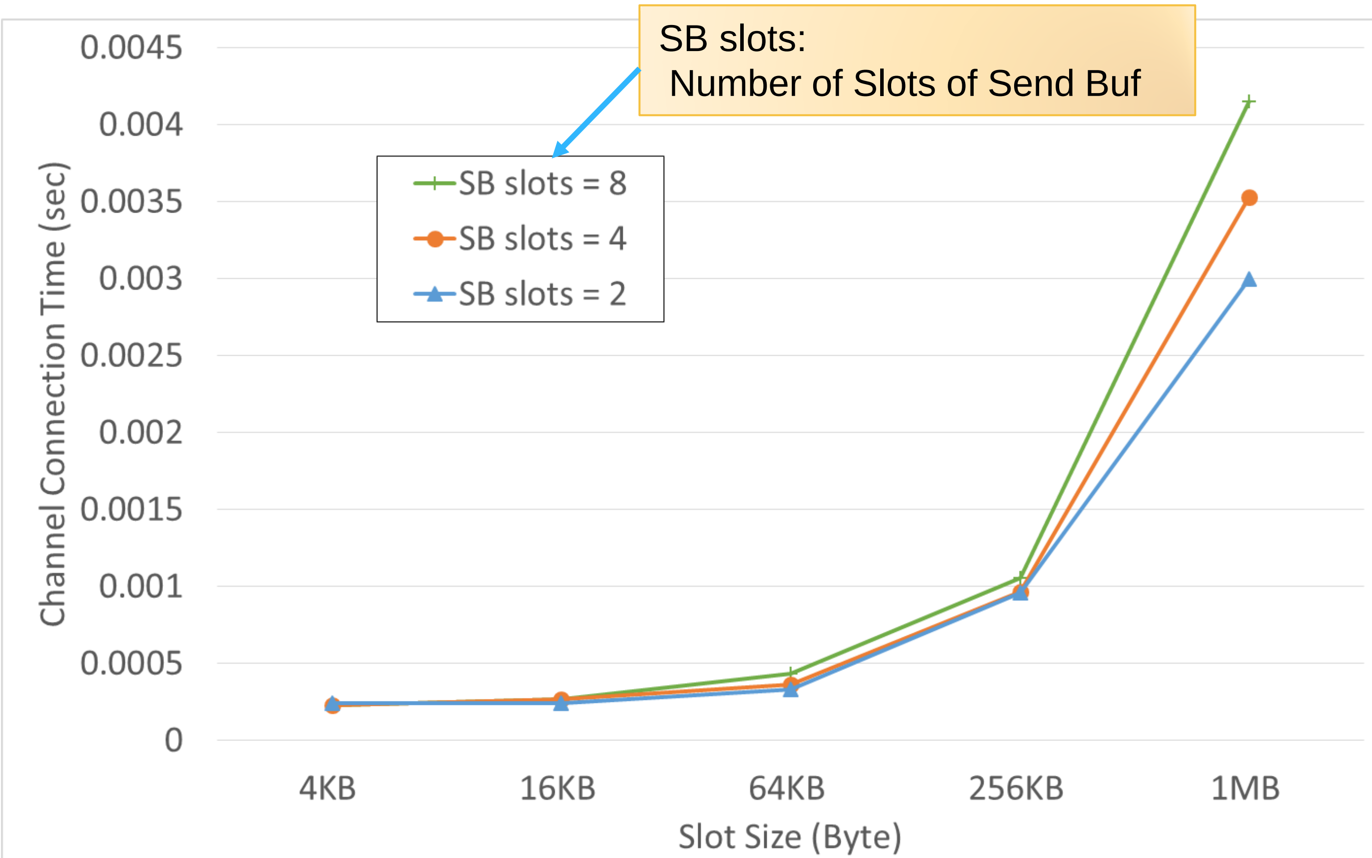
Conclusion and Future Work

- Proposed a set of interfaces for
  Message Passing with User Controlled Connection

- Future Work:
  - More interfaces
  - Build MPI_Send/Recv over Channel Interface

Thank you for your attention.

Performance: Time for Connection

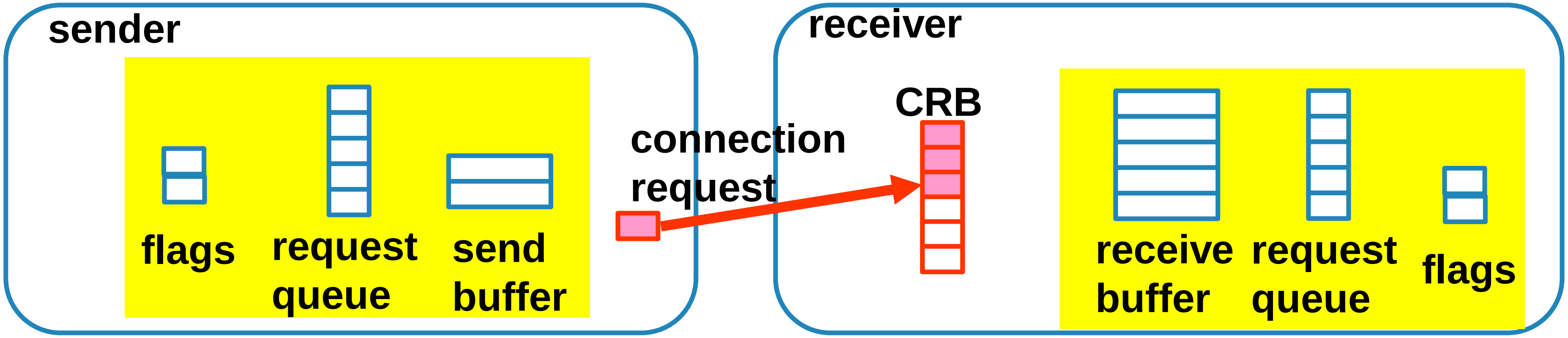


Environment:
PC Cluster (Xeon E5-2609, 8GB RAM, InfiniBand QDR)

Implementation: Channel Creation

- At the initialization:
  Prepare CRB (Connection Request Buffer) on each process for accepting connection requests.

- Creation of a channel:
  1. Allocate data structures on both sides of the channel
     (Message buffer, Request queue, Flags)
  2. The sender exclusively adds a connection request
     to CRB of the receiver.
  3. The receiver matches it with the request of itself and replies Ack.



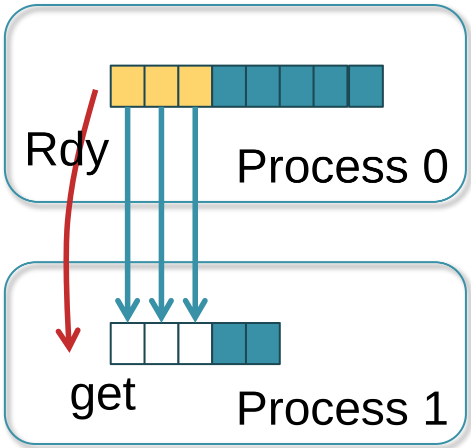
BoF 15: Application development using the ACP library

Toshiya Takami (Kyushu Univ.)

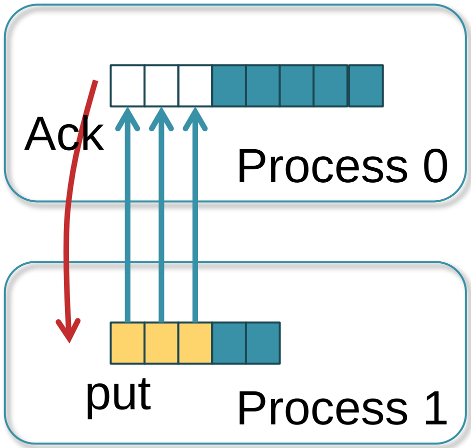
ISC2015
15 July, 2015
14:15 – 15:15

How to use memory efficient communications

- ACP Basic Layer
  - uses Global Memory Access
  - enables us to configure
    - Peer-to-peer communications
    - One-sided communications
    - Data structures



- Production/consumption of data
  - synchronization
    - requires extra control packets for Rdy/Ack
  - to achieve high performance
    - tuning for specific communication patterns



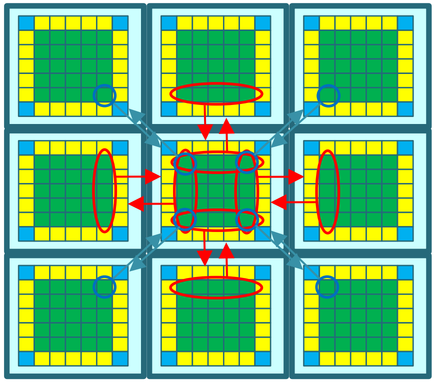
Examples

- Message Passing ←→ PGAS-like Description
  - Halo communications and calculations in stencil
  - Master-worker: channels ←→ global counter

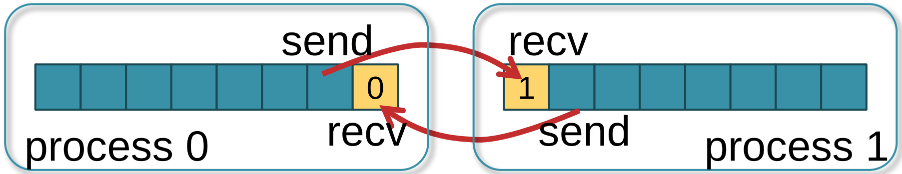
- One-sided Access to Global Memory
  - SpMV by minimized data transfer: get
  - Global Data Structure Library: asynchronous put / get
  - Partcles on adaptive mesh: channel

Halo Communications in Stencil

```
time-step loop {
    exchange_halo_nb();
    calc_innerArea();
    wait();
    calc_halo();
}
```



- Message Passing:
  - send / recv



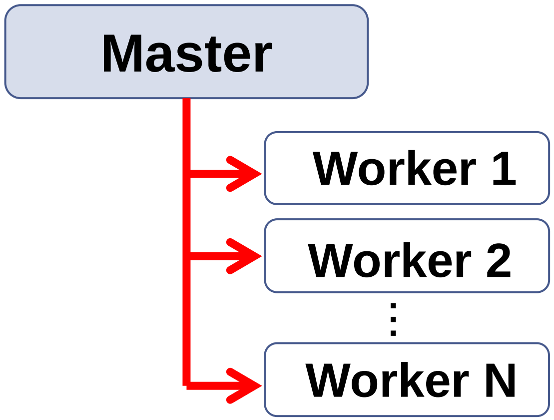
- Global Memory Access
  - copy / completion

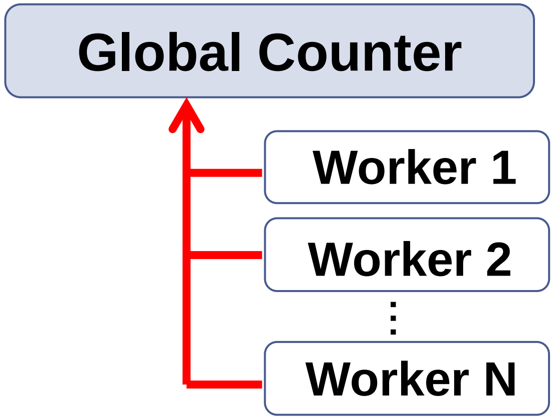
process 0

```
exchange_GA();
...
time-step loop {
  acp_copy(1);
  calc_innerArea();
  acp_complete(1);
  calc_halo();
}
```

Master-Worker

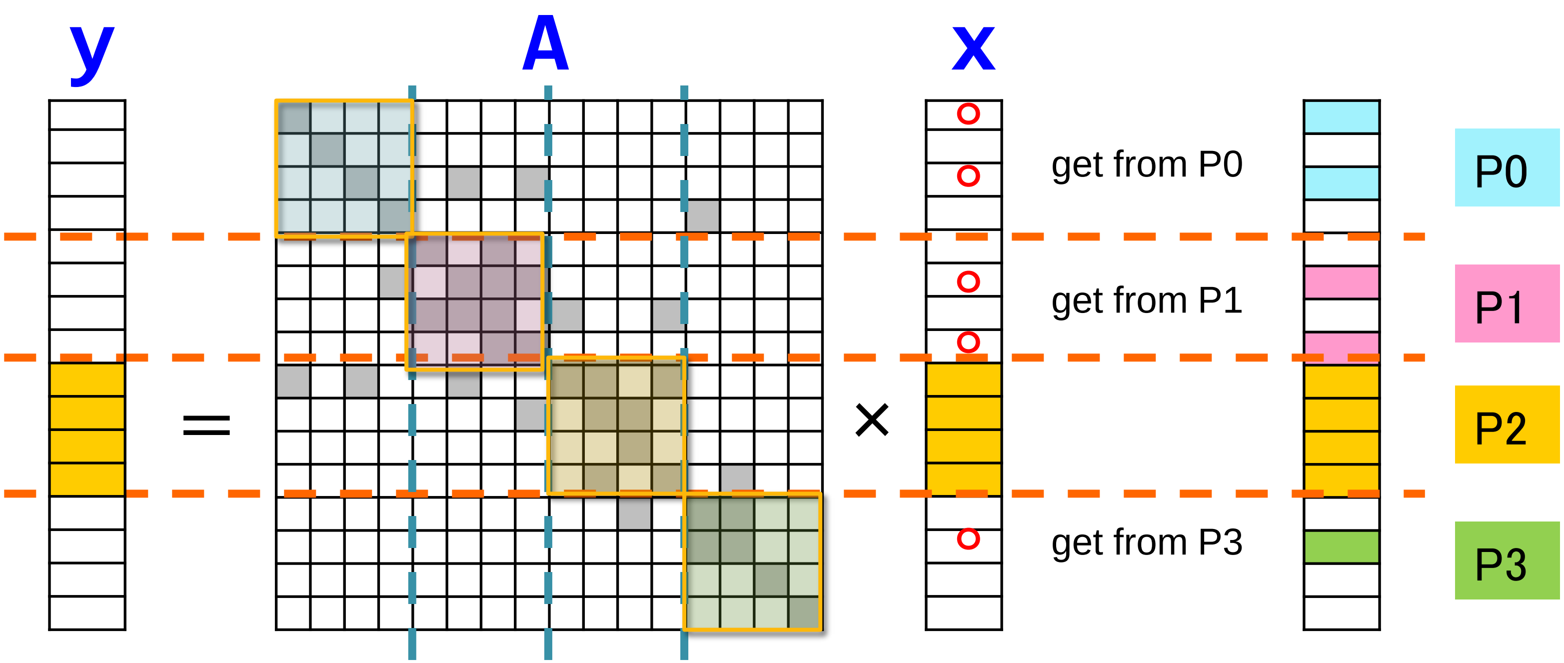
- Master-responsible configuration
  - Master distributes tasks to workers 1-N
  - Releasing channel connections enables us to reduce memory consumption
- Global Counter
  - is implemented by atomic-fetch-and-add
  - represents a pointer to task chunks
- Voluntary-worker configuration
  - is introduced by the Global Counter
  - After incrementing the counter, workers take tasks voluntarily





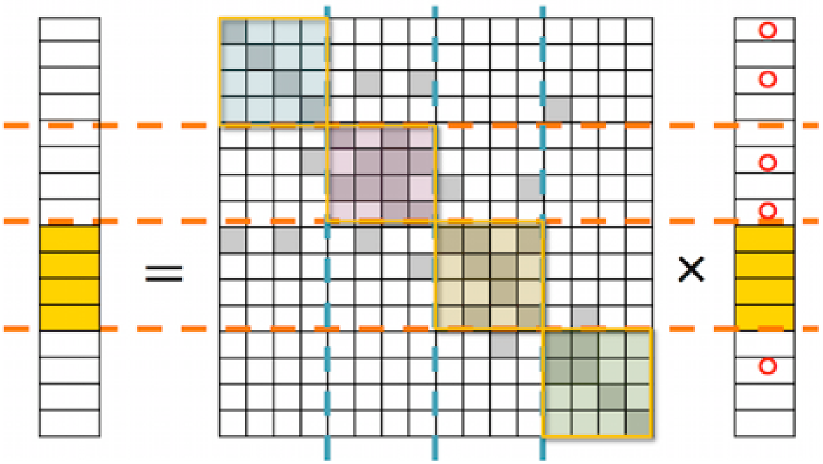
SpMV by minimized data transfer (1)

- Matrix-vector multiplication: $\boldsymbol{y} = \boldsymbol{Ax}$
  - $\boldsymbol{A}$: a sparse matrix (real, non-symmetric): row-block distributed
  - $\boldsymbol{x}, \boldsymbol{y}$: real vectors: block distributed
- Main calculation is in diagonal blocks.
- Data transfer depending on distribution of non-zero elements
  - is naturally written by one-sided accesses.



SpMV by minimized data transfer (2)

```
exchange Global Addresses
...
CG iteration {
    get elements in off-diagonal
    MxV in a diagonal block
    wait until the transfer is completed
    MxV in off-diagonal block
    barrier to calculate dot products
}
```

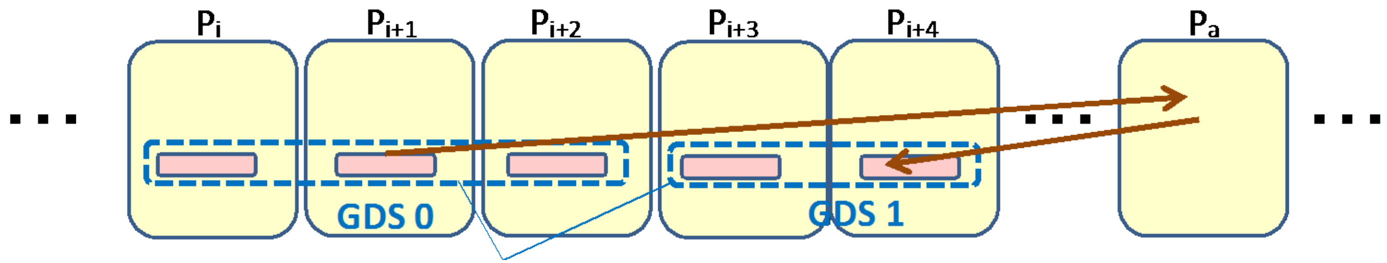


- Latency of the get operation
  - is concealed behind the main calculation for the diagonal block
- Dynamic load-balancing
  - requires another type of irregular communications

Global Data Structure Library

- Global Data Structure
  - is written by vector interfaces in the ACP data library
  - enables irregular/asynchronous accesses from all processes through read (get) / write (put) operations
  - is applied to density-matrices in quantum chemistry calculations

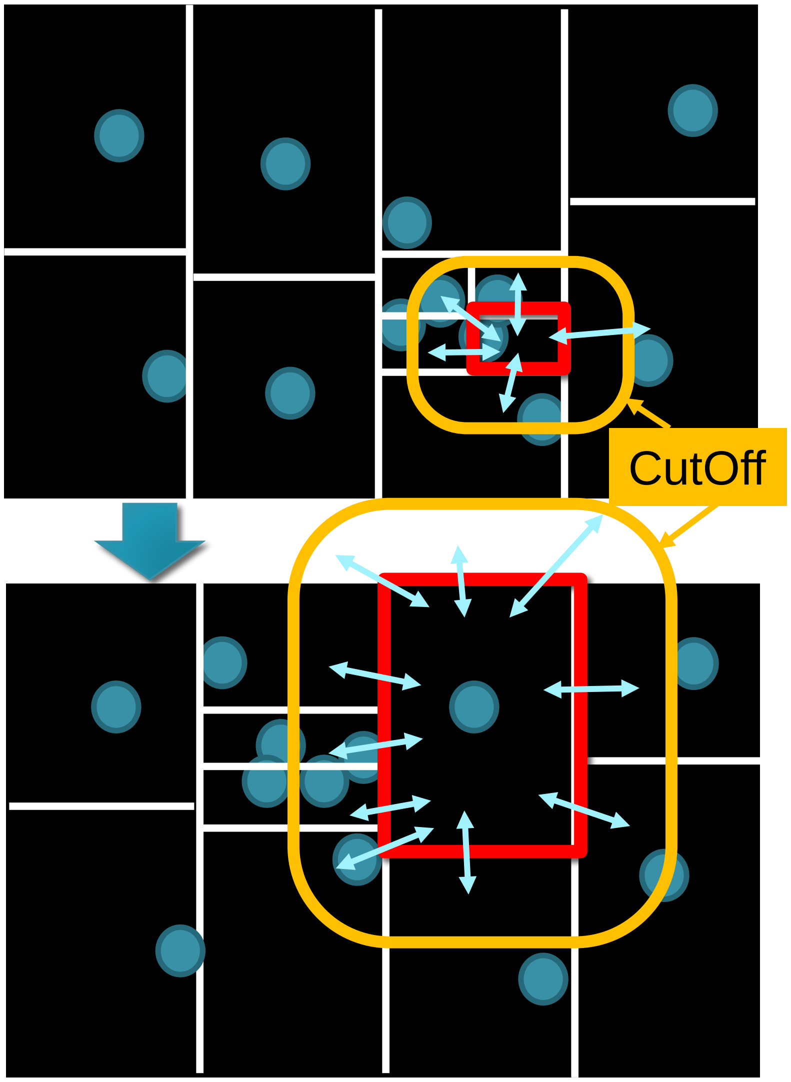
```
gds = acp_create_gds( size );                    Create shared data area
acp_write_gds( gds, offset, size, data );        Write to data area
acp_free_gds( gds );                             Free data area
```



- Data areas accessed from all processes
- Large data is distributed over several processes
- Similar to file I/O interfaces
- Zero copy communication

Particles on Adaptive Mesh

- Non-uniform Particle Distribution
  - leads to unstructured partitioning
  - require irregular communications within a cut-off radius
- When particles move over cells
  - cell partitioning should be modified
  - adjacent pairs change in time
- Memory efficient communication
  - is achieved by releasing resources related to unnecessary connections



Conclusion

- Applications written in message passing
  - Standard stencil codes are also written with ACP library
  - Master-worker pattern is effectively described by a global counter

- Some applications using one-sided access
  - are naturally written by ACP communication/data libraries
  - achieve both of high performance and memory efficiency