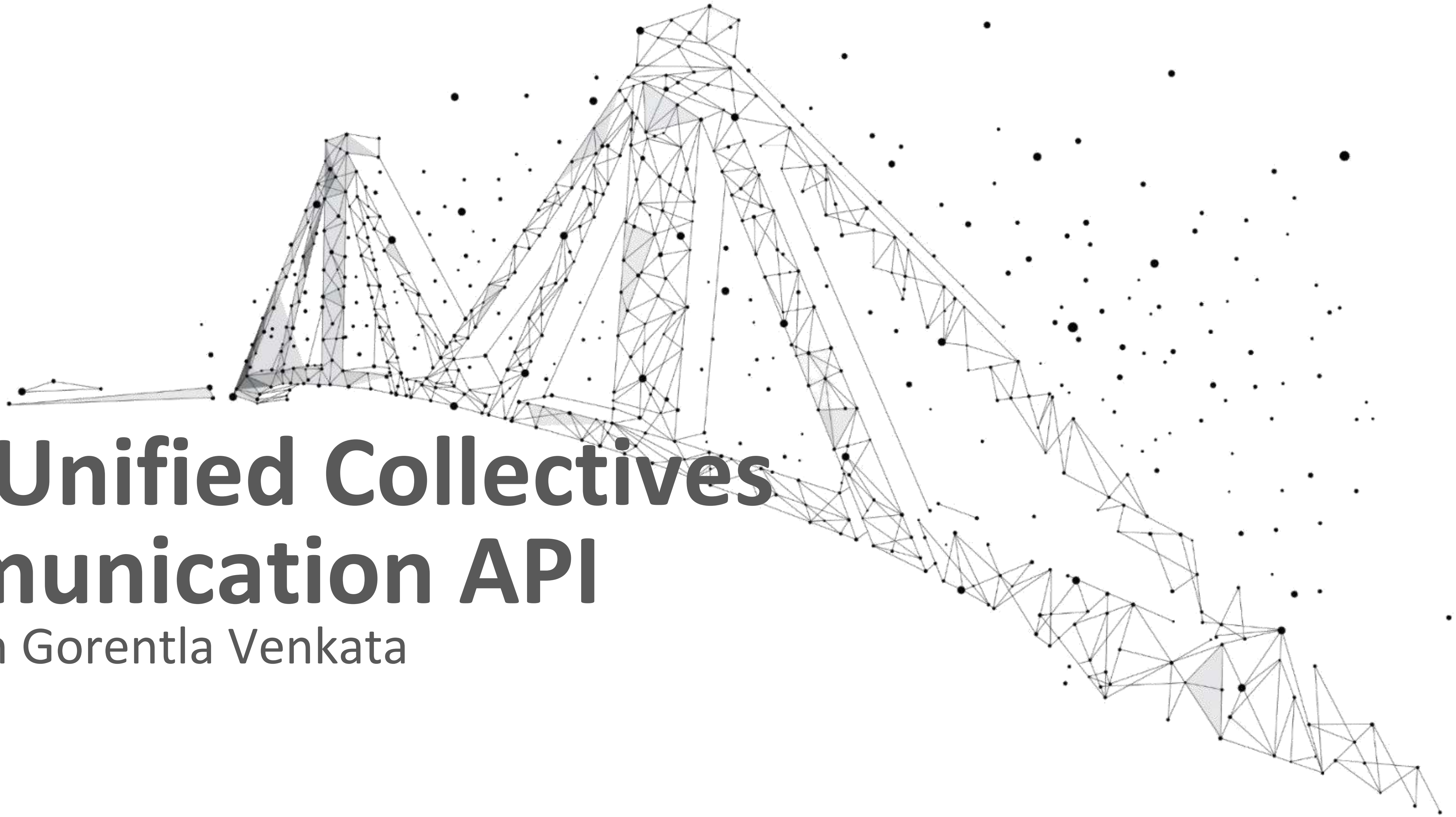


UCC: Unified Collectives Communication API

Manjunath Gorentla Venkata

UCF F2F

December 2019

How to read this presentation ?

- Presentation introduces the abstraction, concepts, and semantics
  - Interfaces, structures, and library constant details are in the API document
- Focus on the big picture for this presentation
  - Details can be debated
- Do not focus on naming, yet
  - We can change the names later. For example, a ***team*** can be named as ***group*** or ***communicator***

UCC: Unified Collective Communication Library

Proposal : Collective communication operations API that is flexible, complete, and feature-rich for current and emerging programming models and runtimes.

High-level Features

- Blocking and Nonblocking collective operations
- Hierarchical collectives are a first-class citizen
  - Well-established design for achieving performance and scalability
- Hardware collectives are a first-class citizen
  - Well-established model and have demonstrated to achieve performance and scalability
- Flexible resource allocation model
  - Support for lazy, local and global resource allocation decisions
- Support for relaxed ordering model
  - For AI/ML application domains
- Flexible synchronous model
  - Highly synchronized collective operations (MPI model)
  - Less synchronized collective operations (OpenSHMEM and PGAS model)
- Repetitive collective operations (init once and invoke multiple times)
  - AI/ML collective applications, persistent collectives
- Point-to-point operations in the context of group
- Global memory management
  - OpenSHMEM PGAS, MPI, and CORAL2 (RFP)

Key Abstractions : Overview

Design around simple set of key abstractions for flexibility and efficiency

- **Communication (Team) Library:** An abstract object representing the library
- **Communication Context:** Encapsulates local resources and topology for group operations.
- **Team:** Encapsulates global resources and team members for group operations.
- **Endpoints:** Encapsulates the members of the ***team***
- **Collective Operation:** Represents the collective operation
- **Task and task list:** Represents groups of collectives

Key Abstractions

1. Communication (Team) Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operation
6. Task and task list

Library : Initialize and finalize

```
ucc_team_lib_init(ucc_lib_team_params_t ucc_params, ucc_team_lib_t *team_lib);

ucc_team_lib_finalize( ucc_team_lib_t team_lib);
```

Semantics:

- Library initialization and finalization allocate and release resources
- All library resources are created and finalized during/after the initialization and finalization calls respectively
  - No operations on the library are valid after the finalize operation
  - No overlapping of Init and finalize call (i.e., Init – Init – Finalize – Finalize on a single thread is invalid behavior)
- The library can be coupled with UCX (UCP context) during initialization
- The library can be customized for a specific programming model

Key Abstractions

1. Communication (Team) Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operation
6. Groups of Collectives

Communication Context (1)

An object to encapsulate local resource and express network parallelism

```
ucc_create_team_context(ucc_team_lib_t comm_lib_context, ucc_team_context_config_t
ctx_config, ucc_team_context_t *comm_context);

ucx_destroy_team_context(ucc_team_context_t team_context);
```

Semantics:

- Context is created by ***ucc _create_team_context(),*** a local operation
- Contexts represents a local resource - threads, injection queue, and/or network parallelism
  - Example: software injection queues (UCP Worker, List of UCP Endpoints), Switch local resources, Hardware injection resources
- Context can be coupled with threads, processes or tasks
  - A single MPI process can have multiple contexts
  - A single thread (pthread or OMP thread) can be coupled with multiple contexts

Communication Context (2)

An object to encapsulate local resource and express network parallelism

```
ucc_create_team_context(ucc_team_lib_t comm_lib_context, ucc_team_context_config_t
ctx_config, ucc_team_context_t *comm_context);

ucx_destroy_team_context(ucc_team_context_t team_context);
```

Semantics:

- Context can be bound to a specific core, socket, or an accelerator
  - Provides an ability to express affinity
- Context can participate in multiple group operations
  - Private context can participate in only one group operation (team)
  - Shared context can participate in multiple group operations
- Multiple contexts per team (from same thread) can be supported
  - Software and hardware collectives

Customizing Context

The usage model, operations supported, thread model, and invocation/completion can be customized.

```
struct ucc_team_context_config {
	ucc_network_ops_t ops;
	ucc_threading_support_t thread_support;
	ucc_team_completion_type_t  completion_type;
	ucc_team_usage_type_t usage;
}
```


Customizing Context: Usage Model

Options:

- UCC as Network Library
  - User implements the collective algorithms and UCC implements the data transfer channels in the context of team
- UCC as Collective library
  - UCC implements the collective algorithms and data transfer channels

Use cases:

- Require collective algorithms and implementation for collective communication
  - Programming models using UCX for point-to-point communication
- Require a thin abstraction over hardware collective primitives
  - Collective libraries that have explored and implemented collective algorithms
- Require a thin abstraction over point-to-point operations and need group abstractions
  - OpenSHMEM contexts
  - MPI Windows

Customizing Context: Operations Supported

Helps with transport selection, resource allocation, and management

Options:

- Only Point-to-point operations
  - Enables creation of resources for only RMA and Point-to-point operations
- Only Collective operations
  - Enables creation of resources for only collective operations
- No communication operation
  - Enables creation of group but no resources are allocated for collectives or RMA/P2P operations
  - Use case: Required for symmetric memory APIs, Memory allocation routines in OpenSHMEM
- Both Point-to-point and collective communication operations are supported

Customizing Context : Threads and Contexts

Provides well-defined interaction between the threads and local resources

- Provide options for performance, flexibility and resource usage
- Sharing of resources between Teams

Options:

- SINGLE
  - The context is accessed by a single thread
  - The context participates in a single Team
    - So resources are exclusive to one Team
  - The libraries can implement it as a lock-free implementation

- SHARED
  - The context is accessed by multiple threads
  - The context can participate in multiple teams
    - Resources are shared by multiple teams
  - The library is required to protect critical sections

Customizing Context: Invocation and Completion

Options:

- Blocking: All operations on the context are blocking
- Non-blocking: All operations on the context are non-blocking operations
- Split-phase: One outstanding operation at a time, however, completion can be delayed
- Default: Both blocking and non-blocking operations can be posted

Use cases:

- OpenSHMEM only supports blocking operations.
- Support for split-phase barriers
- Support for persistent collective semantics

Key Abstractions

1. Communication (Team) Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operation
6. Groups of Collectives

Team: Membership

Who manages the participation in the group?

User Managed

- The user manages who participates in the team
  - The user provides an OOB collective operation to exchange ***context*** among the members
  - The members join the collective operation
  - The scope of the team is defined by the OOB collectives
    - For example, if the OOB is defined over shared memory, the team is created over shared memory.
    - **"UCC_TEAM_WORLD"** is created by using PMIx collectives as OOB collectives

Library managed:

- The library (UCC) manages the membership
  - UCC performs and implement a collective operation to determine the participation

Team: Operations for creating teams

```
ucc_team_create_post(
        ucc_team_context_t  team_context, ucc_team_config_t comm_config, oob_collectives_t
oob_collectives, ucc_team_ep_t *my_ep, ucc_team_t *new_team);

ucc_team_create_wait();
```

Semantics:

- Created by processes, threads or tasks by calling ***ucc_team_create_post()***
  - A collective operation but no explicit synchronization among the processes or threads
- Non-blocking operation and only one active call at any given instance.
- Each process or thread passes local resource object ***(context)***
  - Achieve global agreement during the create operation
- Passing NULL as OOB will result in creating a "world" team

- Create global resources for group communication buffers
  - Synchronization buffers for one-sided collectives
  - Temporary buffers for reduction operations
  - Scratch buffers for non-blocking operations
  - Create connections if required
  - Filter the available operations and algorithms

Team : Customizing team

```
struct ucc_team_config_t {
        ucc_post_ordering ordering;
        uint64_t num_outstanding_collectives;
        ucc_team_completion_type_t  comple
        tion_type;
        ucc_collective_sync_type_t sync;
        ucc_ep_range_contig ep_range;
        ucc_dt_type_t datatype;
        ucc_mem_params_t mem_params;
        }
```

Semantics:

- Ordering : All team members must invoke collective in the same order?
  - Yes for MPI and No for TensorFlow and Persistent collectives
- Outstanding collectives
  - Can help with resource management
- Blocking/Non-blocking
  - A ***team*** can be customized to be either blocking or non-blocking
- Should Endpoints in a contiguous range ?
- Datatype
  - Can be customized for contiguous, strided, or non-contiguous datatypes
- Synchronization Model
  - On_Entry, On_Exit, or On_Both – this helps with global resource allocation

Customizing Team: Synchronizing Model

- **NO_SYNC_ON_Entry:** No synchronization on entry
  - On entry each process can start the collective irrespective of other processes entered the collective or not
  - Data readiness is ensured by the programming model user (not programming model itself)
  - Use case : OpenSHMEM / UPC

- **NO_SYNC_ON _Exit:** No synchronization on exit
  - On exit each process can exit the collective irrespective of other processes completed or not
    - Provides guarantees about local completeness, not global state
  - Use case/ Motivation: Broadcast, OpenSHMEM / UPC

- **NO_SYNC:** No synchronization on entry or exit
  - Data readiness is ensured by the User
  - Global completion guarantees are to be learned by the user
  - Use case : OpenSHMEM/UPC

- **Default:** Synchronization on both entry and exit to the collective
  - Data readiness is ensured by the programming model and provides global state on completion

Team : Query Operations

```
ucc_get_team_attribs(ucc_team_t ucc_team, ucc_team_attrib_t *team_atrib)
ucc_get_team_size(ucc_team_t ucc_team);
ucc_get_team_my_ep(ucc_team_t ucc_team, ucc_team_ep_t *ep);
ucc_get_team_all_eps(ucc_team_t ucc_team, ucc_team_ep_t *ep, uint64_t num_eps);
```

- All attributes of the ***team*** are available via ***ucc_team_attrib_t***
  - Size, ordering, sync type, completion semantics, datatype, endpoints, and memory handles
- Interfaces for some common attributes
  - Size and Endpoints

Team : Splitting teams

Supporting split operations in lower libraries will enable resource sharing between parent and child teams

```
ucc_team_create_from_parent( ucc_team_ep my_ep,  int color,  ucc_team_t parent_team,
ucc_team_t *new_ucc_team);
```

Semantics:

- Split
  - Collective operation over the parent team
  - Collective operations over the child team or can be a local operation (interface in the later slides)
- Provides flexible way to create a team
  - Supports regular as well as irregular team creation
- Inherits configuration from the parent team
- Thread model: One active split operation per process

Key Abstractions

1. Communication (Team) Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operation
6. Groups of Collectives

Endpoint

An integer that represents the network address and/or team member

```
ucc_create_team_from_ep_list(ucc_team_t parent_ucc_team, ucc_team_ep *ep, uint64_t num_eps, ucc_team_t *new_team);

ucc_create_team_from_ep_stride(ucc_team_t parent_ucc_team, uint64_t start_ep, uint64_t stride, uint64_t num_eps, ucc_team_t *new_team);

ucc_team_add_endpoint(ucc_team_t parent_ucc_team,  ucc_team_context_t *team_context, ucc_team_ep ep, ucc_team_t *new_team);
```

Use case:

- Team creation only with a collective operation on the newly created team
- Light-weight team creation by passing the list of endpoints
  - Enables lazy resource allocation
- Support spawn semantics .i.e., supports adding an endpoint to the team

Key Abstractions

1. Communication (Team) Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operations
6. Task and task list

Collective Operations : Building blocks (1)

```
ucc_collective_init( ucc_coll_op_args coll_args, ucc_team_t team, ucc_coll_op_h *coll_handle);
ucc_collective_init_and_post( ucc_coll_op_args coll_args, ucc_team_t team, ucc_coll_req
*request, ucc_coll_op_h *coll_handle);

int ucx_collective_post(ucc_coll_op_h coll_handle, ucc_coll_req *request)
int ucx_collective_test(ucc_coll_req request);
int ucx_collective_wait(ucc_coll_req request);
int ucx_collective_finalize(ucc_coll_req request);
```

Collective Operations : Building blocks (2)

Semantics:

- Collective operations : ***ucc_collective_init(…)*** and ***ucc_collective_init_and_post(…)***
- Local operations: ***ucc_collective_post, test, wait, finalize***
- Initialize with ***ucc_collective_init(…)***
  - Initializes the resources required for a particular collective operation, but does not post the operation
- Completion
  - The ***test*** routine provides the status, and ***wait*** routine can be used to complete the operation
- Finalize
  - Releases the resources for the collective operation represented by the request
  - The post, test, and wait operations are invalid after finalize

Collective Operations : How to build various collectives ?

```
ucc_collective_init( ucc_coll_op_args *coll_args, ucc_team_t team, ucc_coll_op_h *coll_handle);
ucc_collective_init_and_post( ucc_coll_op_args *coll_args, ucc_team_t team, ucc_coll_req *request,
ucc_coll_op_h *coll_handle);

int ucx_collective_post(ucc_coll_op_h *coll_handle, ucc_coll_req *request)
int ucx_collective_test(ucc_coll_req request);
int ucx_collective_wait(ucc_coll_req request);
int ucx_collective_finalize(ucc_coll_req request);
int ucx_collective_req_status(ucc_coll_req request);
```

- Nonblocking and blocking collectives:
  - Can be implemented with Init_and_post and wait+finalize
- Persistent Collectives:
  - Can be implemented using the building blocks - init, post, test, wait, finalize
- Split-Phase
  - Can be implemented with Init_and_post and wait+finalize

Customizing Collective Operation (1)

```
typedef struct ucc_collective_op_arguments
{
        ucc_collective_type coll_type;
        ucc_coll_buffer_info_t buffer_info;
        ucc_collective_sync_type_t sync_type;
        ucc_reduction_op reduction_info;
        ucc_error_type_t error_type;
        ucc_coll_tag_t coll_id;
        ucc_team_endpoint_t root;
} ucc_coll_op_args;
```

- Collective type, buffer information, and reduction info
  - Customize the operation
- Synchronization type
  - Same sync_type as context_config / comm_config.
  - Valid to use the default (all synchronization) even when context and config are configured as on_entry, on_exit, or on_both but not vice versa
- Collective Tag
  - For unordered collectives
- Root endpoint for root-based operations

Customizing Collective Operation (2)

Operation and Reduction Types

```
enum ucc_collective_type {
        Barrier,
        Alltoall,
        Alltoallv,
        Broadcast,
        Gather,
        Allgather,
        Reduce,
        Allreduce,
        Scatter,
        FAN_IN,
        FAN_OUT
}
```

```
enum ucc_reduction_op {
        OP_MAX,
        OP_MIN,
        OP_SUM,
        OP_PROD,
        OP_AND,
        OP_OR,
        OP_XOR,
        OP_MAXLOC,
        OP_MINLOC
}
```

Customizing Collective Operation (3)

Buffer Information

```
typedef struct ucc_coll_buffer_info {
        void *src_buffer;
        size_t src_len;
        void *dest_buffer;
        size_t dest_len,
        int64 flags, /* in-buffer */
} ucc_coll_buffer_info_t
```

- src_buffer, src_len, dest_buffer, and dest_len standard semantics
- Flags
  - Persistent
  - Symmetric
  - In-buffer

Customizing Collective Operation (3)

Error Types

```
enum ucc_error_type {
        LOCAL=0,
        GLOBAL=1,
}
```

- **Local:**
  - There is no agreement on the errors reported to the members
  - If agreement is needed, it is the user responsibility to achieve it
- **Global:**
  - All members return the same error

Customize Context or Team or Collective Operation?

This is a philosophical question as it varies with the programming environment. So, some guidelines

- Make a local decision, when you can.
  - This reduces the number of global decisions, hence fewer collectives during initialization
  - Can change the decision with less cost. i.e., no collective required
- Provide mechanism to modify local decision during the global agreement process
- Provide mechanism to modify the local decision or global decision during the invocation time

Key Abstractions

1. Communication (Team) Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operations
6. Task and task list

Collective Groups

Collective groups are a group of ordered or un-ordered collective operations

Use Case:

- Collective groups enable the implementation of hierarchical collectives
  - It is well established that by tailoring the algorithm and customizing the implementation to various communication mechanisms in the system can achieve higher performance and scalability

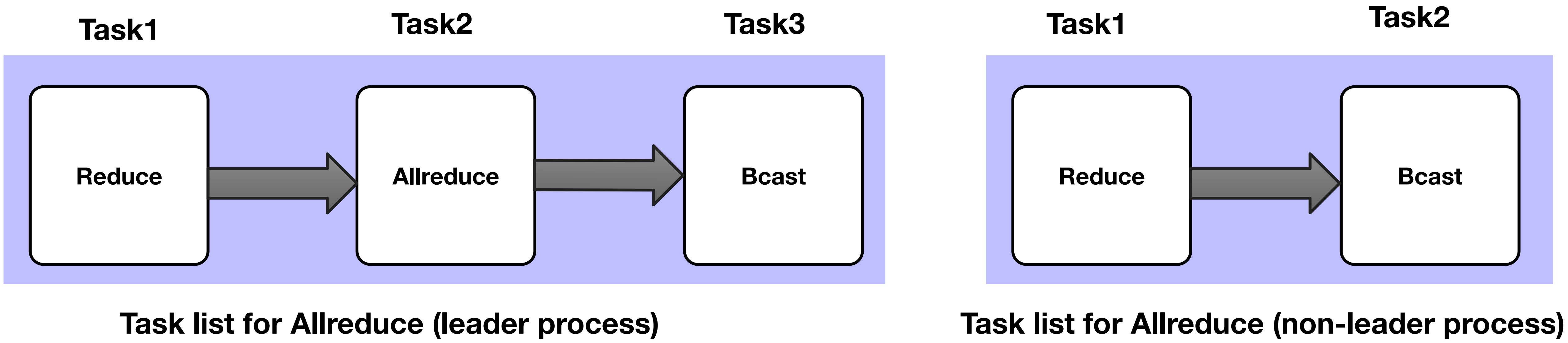
How to express groups of collectives?

- Triggered Operations
  - o Pros: Hardware Support
  - o Cons: Expressing
- Collective Schedules as DAGs
  - o Pros: Highly Expressible (parallelism, dependencies)
  - o Cons: Leveraging hardware trigger mechanism is tricky
- Chained/List Collective Operations
  - o Pros: Easy to program and implement
  - o Cons: Expressing parallelism can be a bit awkward

Collective Groups: Task and Task List

Collective groups are a group of ordered or un-ordered collective operations

- **Task:** Represents a collective operation and its corresponding team
- **Task List:** Represents a collective operation group executed either in order or unordered



Collective Groups

Operations to create and execute tasks

```
ucc_create_coll_task(ucc_coll_op_args_t args, ucc_team_t team, ucc_coll_task_t *task);
ucc_create_task_list(int num_tasks, bool ordered, ucc_coll_task_t tasks[], ucc_coll_task_list
*task_list);
ucc_schedule_task_list(int priority, ucc_coll_task_t task_list, ucc_task_execution_t *active_list);
ucc_complete_tasks(ucc_execution_t active_graph);
```

Semantics:

- All task operations are local
- *ucc_create_coll_task()* creates a task from collective arguments and team
- *ucc_create_task_list()* creates either an ordered or unordered list of tasks
- *ucc_schedule_task_list()* schedules the tasks to be executed either parallel(unordered) or serial(if ordered)
  - All members of the team in the task are expected to execute the same collective operation; otherwise, the operation is undefined.
  - All task executions are non-blocking and asynchronous
- *ucc_complete_tasks()* completes the execution of tasks in the task_list

Key Abstractions

1. Communication (Team) Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operations
6. Task and task list

Global memory management

```
ucc_global_mem_alloc(ucc_team_t team, size_t size, ucc_mem_constraints constraints,
ucc_mem_hints hints, ucc_global_mem_t *mem_handle);

ucc_global_mem_free(ucc_global_mem_t mem_handle, ucc_team_t team)

ucc_global_mem_get_attrib(ucc_global_mem_t mem, ucc_global_mem_attrib *attributes);
```

Semantics:

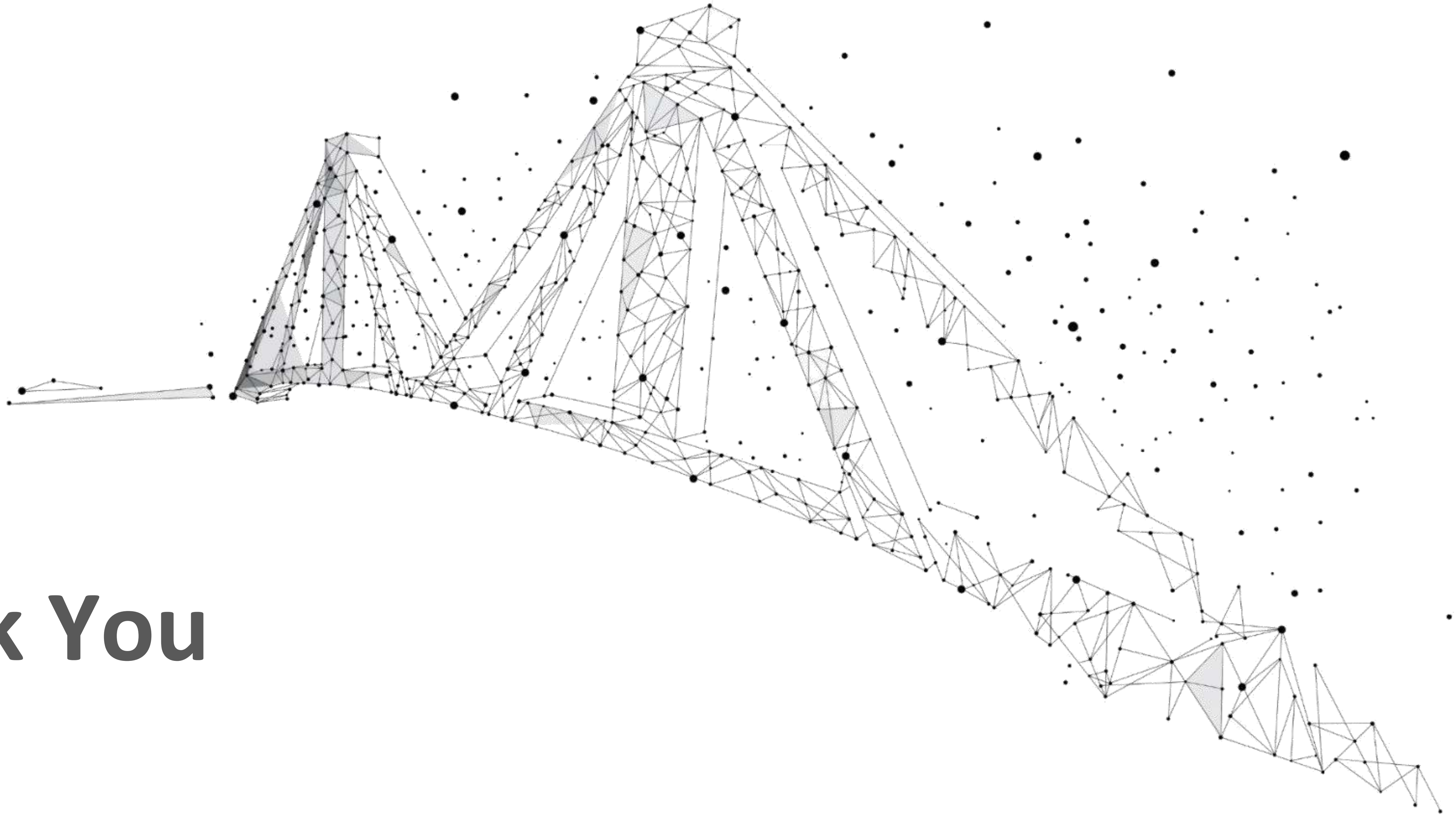
- Manages memory on each of member of the ***team***
- The constraints argument control the semantics
  - Example – symmetric, alignment
- The hints provide information about usage (think about mbind)
  - Memory policy – local, shared,
  - Usage - atomics, counters, small message, large message, MPI windows

Use cases:

- OpenSHMEM heaps, MPI Windows, PGAS models, and requirement for some RFPs (for example CORAL2)
- Internal for collectives – sync buffers, temporary work buffers

A Collective Communication API in UCF should support

- A wider variety of programming models
  - MPI is important for HPC
  - Other programming models are important and will grow in importance
- Hardware collectives should be a first-class citizen
  - Mellanox and other vendors already support hardware collectives
- Hierarchies should be a first-class citizen
  - It is well-established that hierarchical collectives achieve higher performance and scalability
  - UCC API should support abstractions to build hierarchies
- Enable flexible resource allocation
  - Lazy resource allocation
  - Local and global decisions
- Iterative collectives should be supported
  - Build once and invoke multiple times.
- Support for various synchronization models
  - Both strict and relaxed synchronization models should be supported
- Support for P2P operations and global memory allocation operations



Thank You