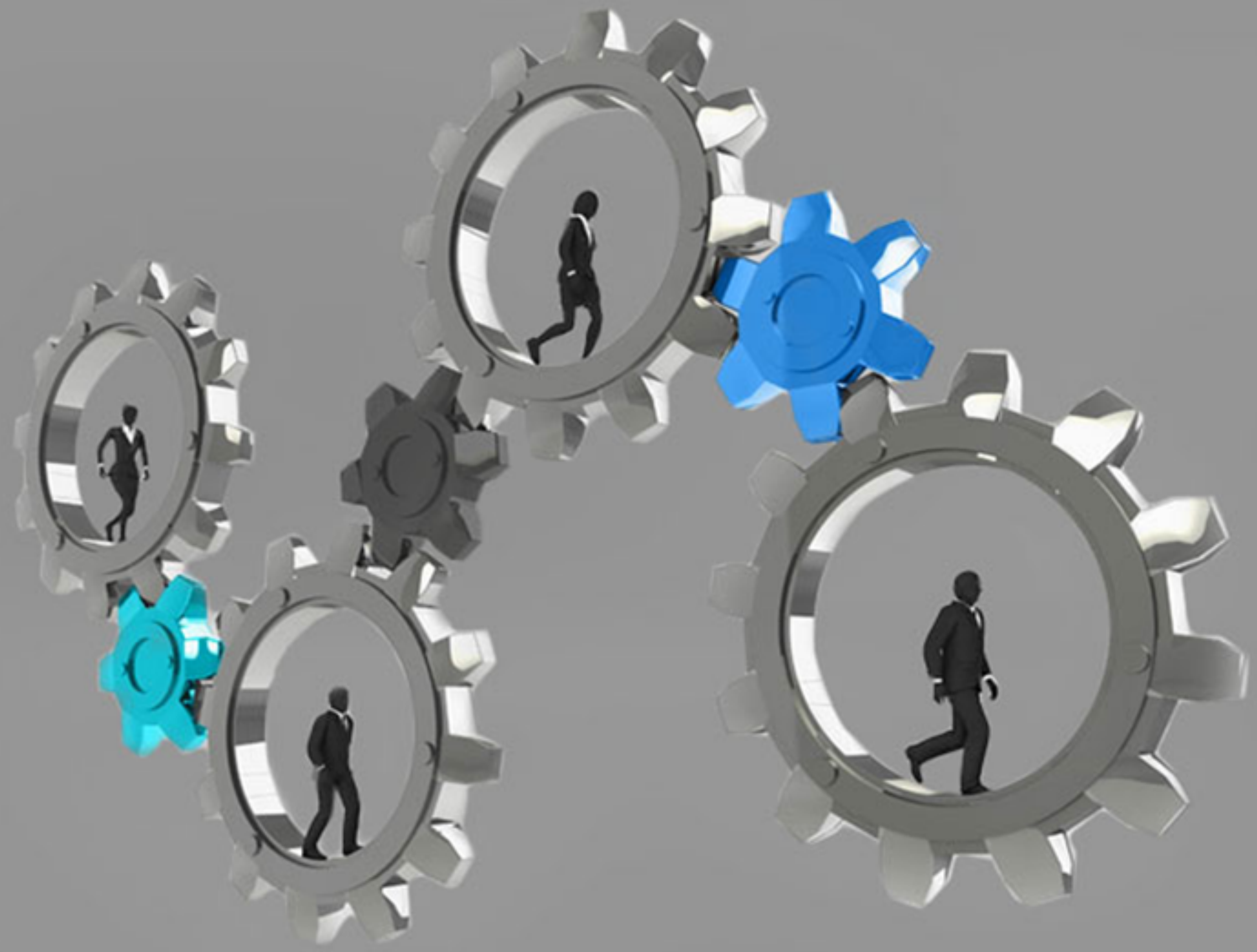
ENABLER OF CO-DESIGN

UCF



UCC API

Manjunath Gorentla Venkata, UCF Collectives WG, Virtual F2F, May 2020

UCC: Unified Collective Communication Library

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

- Highly scalable and performant collectives for HPC, AI/ML and I/O workloads
- Nonblocking collective operations that cover a variety of programming models
- 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
  - 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)

Experimental Implementations Exploring Various Features

- Mellanox's XCCL
- Hierarchical collectives' approach to achieve performance and scalability

https://github.com/openucx/xccl

- Huawei's XUCG
- Reactive based approach

https://github.com/openucx/xucg

- Learn from other implementations
  - PAMI Collectives / IBM libcoll
  - OMPI-X
  - ADAPT
  - HCOLL
  - SHARP hardware

Naming Conventions

- All public functions to be prefixed with ”ucc” and will be defined in “ucc.h”
- All public library constants are prefixed with UCC
- ~~All experimental functions to be prefixed with “xucc”~~

ucc_<class/object>_<action>_<subset>

- Example : ucc_lib_init, ucc_team_create, ucc_team_create_plan
- “create” / “destroy” creates and destroys the objects
- “init” / “finalize” initializes and finalizes the object
- “get” to be used for retrieving object attributes
- “nb” for non-blocking
- “nbr” for non-blocking with request

Abstractions

1. Collective Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operation
6. Task and task list
7. Plan

Library: Initialization and finalize

*ucc_init(ucc_lib_config_t ucc_config, ucc_lib_t *lib_obj);*

ucc_finalize(ucc_lib_t lib_obj);

*ucc_lib_get_attribs(ucc_lib_t ucc_lib, ucc_lib_attrib_t *lib_atrib)*

Library object to be called - ucc_context_t (hold on to this idea for now)

Library: Initialization and finalize

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
- Library initialization is not a collective operation
- ~~No overlapping of Init and finalize call (i.e., Init – Init – Finalize – Finalize on a single thread is invalid behavior)~~
- We want to support the model where multiple Init / Finalize are supported (input from WG – Feb 26th)

Library Initialization: Customizing library

```
typedef struct ucc_lib_config {
    ucc_lib_config_mask_t mask;
    ucc_lib_reproducibility reproducible;
    ucc_lib_thread_mode thread_mode;
    ucc_lib_usage_type_t requested_lib_usage;
    ucc_collective_op_types_t requested_coll_types;
    ucc_reduction_op_t requested_reduction_types;
} ucc_lib_config_t;
```

Library Initialization: Customizing library

```
typedef enum {
        UCC_HW_COLLECTIVES = 0,
        UCC_SW_COLLECTIVES=1
        REACTIVE = 2
        SHARED_MEM = 3
} ucc_lib_usage_type_t;
```

Library Initialization: Collective Operations

- Why do you need this?
  - Provide an interface for Users to convey the required functionality
    - MPI implementations can request only MPI specific collective operations
    - OpenSHMEM implementations (OSHMEM, OSSS-UCX, SOS) can request only OpenSHMEM specific collective implementations
    - AI-specific implementations can request only Reductions, Broadcast, and Barrier implementations
    - OMPI can request required collective operations from UCC and use other non-UCC components
  - Libraries can convey to the User what collectives are implemented.
  - Implementations can tailor the library functionality for the usage scenario (initialize only components)

Customize library for the Use Cases

- MPI - meaning select all collective operations
- OpenSHMEM/UPC - select all collective operations with sync model
- For AI/ML models, we need reductions and broadcast

- Parameters
  - Collective Models – XUCG, XCCL, Hardware, Vendor
  - Collective Operations – Allreduce, Barrier, Alltoall, Gather, Default (all)
  - Synchronization Model - No Sync, Sync , Default (Sync)
  - ~~Priority~~

Library Initialization: Collective Operations

How to express this? What is the right granularity?

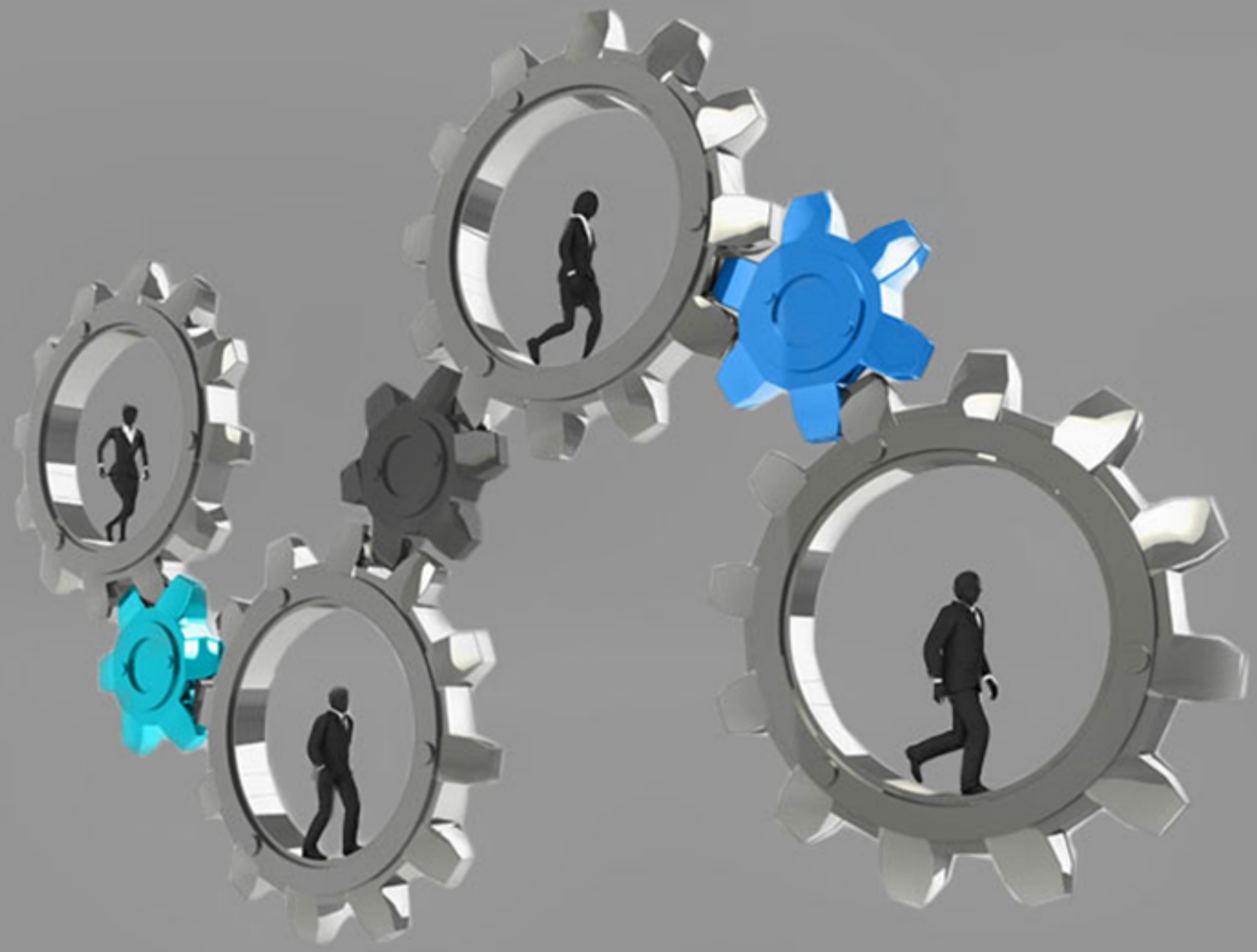
- Coarse-grained: Express at programming model abstraction
  - MPI_MODEL, OPENSHMEM_MODEL, AI_MODEL (Not very standard)
  - Cons: Limited expressibility
- Fine-grained: Express at the fine-grained level of operations, datatype, programming model, ordering
  - Cons: A huge list that might be excessive (not required)
- Strike a balance: Express it as a set of composeable choices
  - Operations – Barrier, Reduce, Alltoall, Alltoallv …
  - Reductions – SUM, PROD, MIN, MAX,
  - Datatypes – Standard datatypes and Extended datatypes
    - Standard datatypes – common set of standard datatypes available in programming models
    - Extended datatypes – user defined datatypes
  - Synchronization Model – Sync and No Sync (Entry and Exit)
  - Ordering Model – Ordered Collectives or Unordered Collectives

Questions

- Should Init/finalize be a collective operation ?
  - No it should be a local function (Feedback from WG Feb 26th)
- How do we handle the race between multiple Init's ?
- Any missing configuration parameters for the initialization ?
- Don't freeze yet, we might require more as we discuss other abstractions
- How do we pass configuration parameters ?
  - 1) Environment variables 2) Configuration files and 3) Interface invocation
  - Support all three options (Feedback from WG March 16th)
  - Add API to read configuration from config files

ENABLER OF CO-DESIGN

UCF



UCC API

Library initialization, local resources abstraction

Manjunath Gorentla Venkata, UCF Collectives WG,

March 25th /April 1st, 2020/April 22nd, 2020

Abstractions

1. Collective Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operation
6. Task and task list
7. Plan

Communication Context (1)

An object to encapsulate local resource and express network parallelism

```
ucc_context_create(ucc_lib_t lib_obj, ucc_context_config_t ctx_config, ucc_context_t *comm_context);

ucc_context_destroy(ucc_context_t comm_context);

ucc_context_get_attrib(ucc_context_t ctx, ucc_context_attrib_t *ctx_attrib);
```

Semantics

- Context is created by ucc_context_create(), a local operation
- Contexts represents a local resource for group operations - injection queue, and/or network parallelism
  - Example: software injection queues (network endpoints), hardware 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_context_create(ucc_lib_t lib_obj, ucc_context_config_t ctx_config, ucc_context_t *comm_context);

ucc_context_destroy(ucc_context_t comm_context);

ucc_context_get_attrib(ucc_context_t ctx, ucc_context_attrib_t *ctx_attrib);
```

Semantics:

- Context can be bound to a specific core, socket, or an accelerator
  - Provides an ability to express affinity
- Context can participate in one or more 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
  - Optimize for bandwidth utilization

Customizing Context

The user can customize synchronization model, usage model, and context types.

```
typedef struct ucc_context_config {
        ucc_context_mask_type_t              mask;
        ucc_context_type_t                   ctx_type_t;
        ucc_context_collective_sync_type_t sync_type;
} ucc_context_config_t;
```

Customizing Context : Context Type

Customize for resource sharing and utilization

EXCLUSIVE

- The context participates in a single team
  - So resources are exclusive to a single team
- The libraries can implement it as a lock-free implementation

SHARED

- The context can participate in multiple teams
  - Resources are shared by multiple teams
- The library might be required to protect critical sections

Customizing Context : Synchronization Models (Updated)

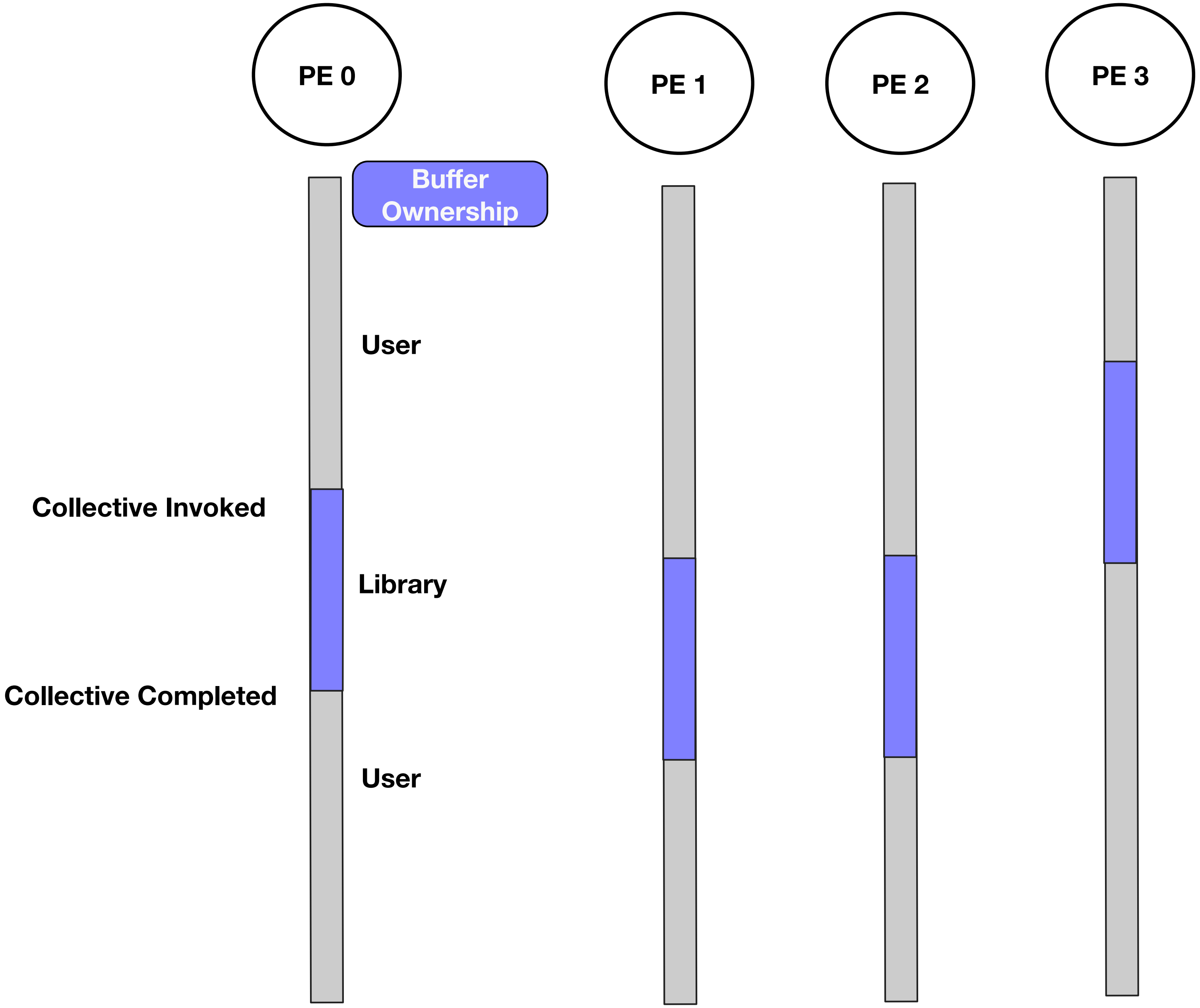
- **NO_SYNC_ON_Entry: No synchronization on entry**
  - On entry, each process/thread can read/write to other processes/threads irrespective of they entered the collective
  - Use case: OpenSHMEM / UPC

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

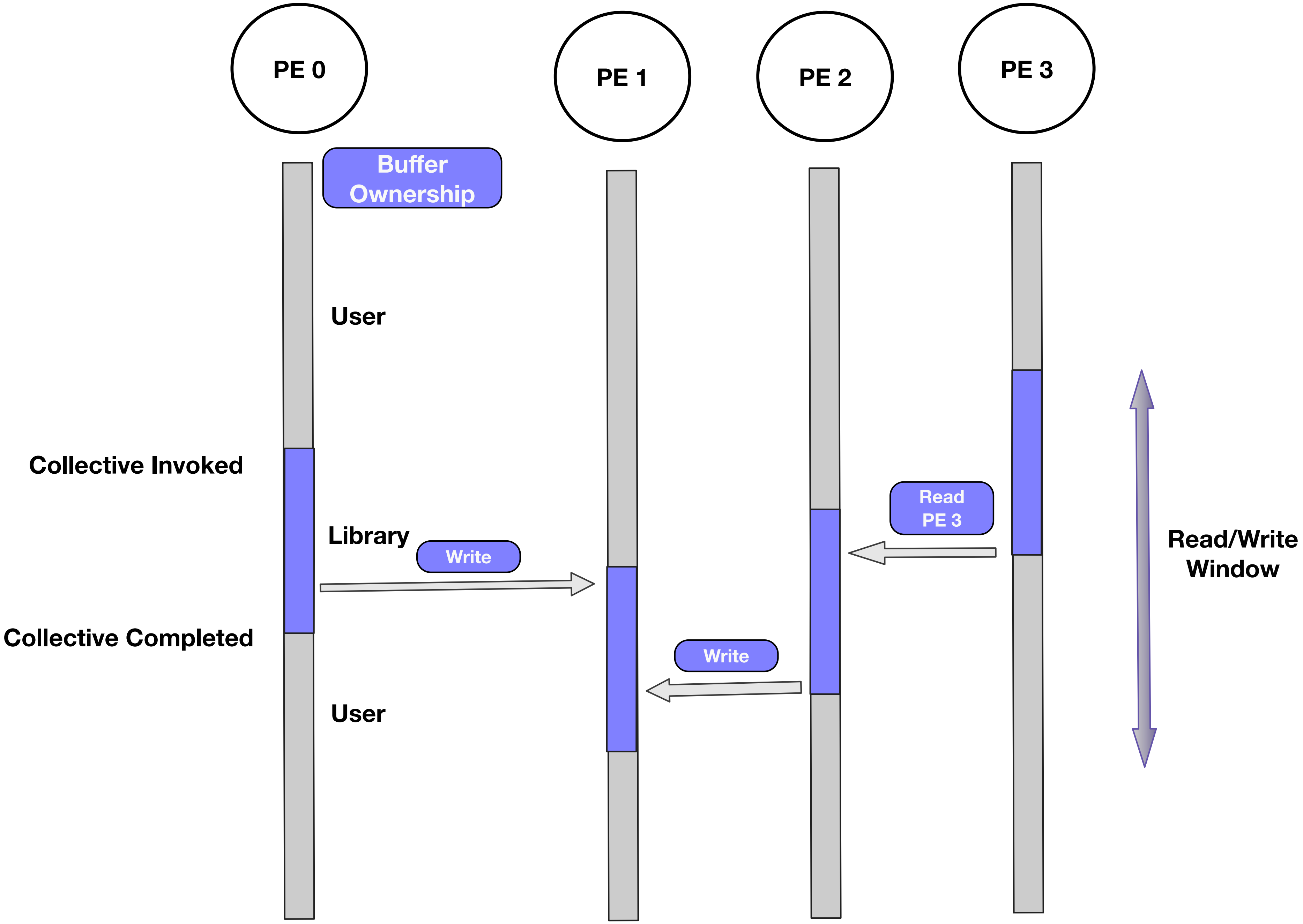
- **NO_SYNC: No synchronization on entry or exit**
  - Can be expressed as **NO_SYNC_ON_Entry | NO_SYNC_ON _Exit**

- **SYNC_ON_BOTH: Synchronization on both entry and exit**
  - On entry, the processes/threads cannot read/write to other processes without ensuring all have entered the collective
  - On exit, the processes/threads may exit after all processes/threads have completed the reading/writing.

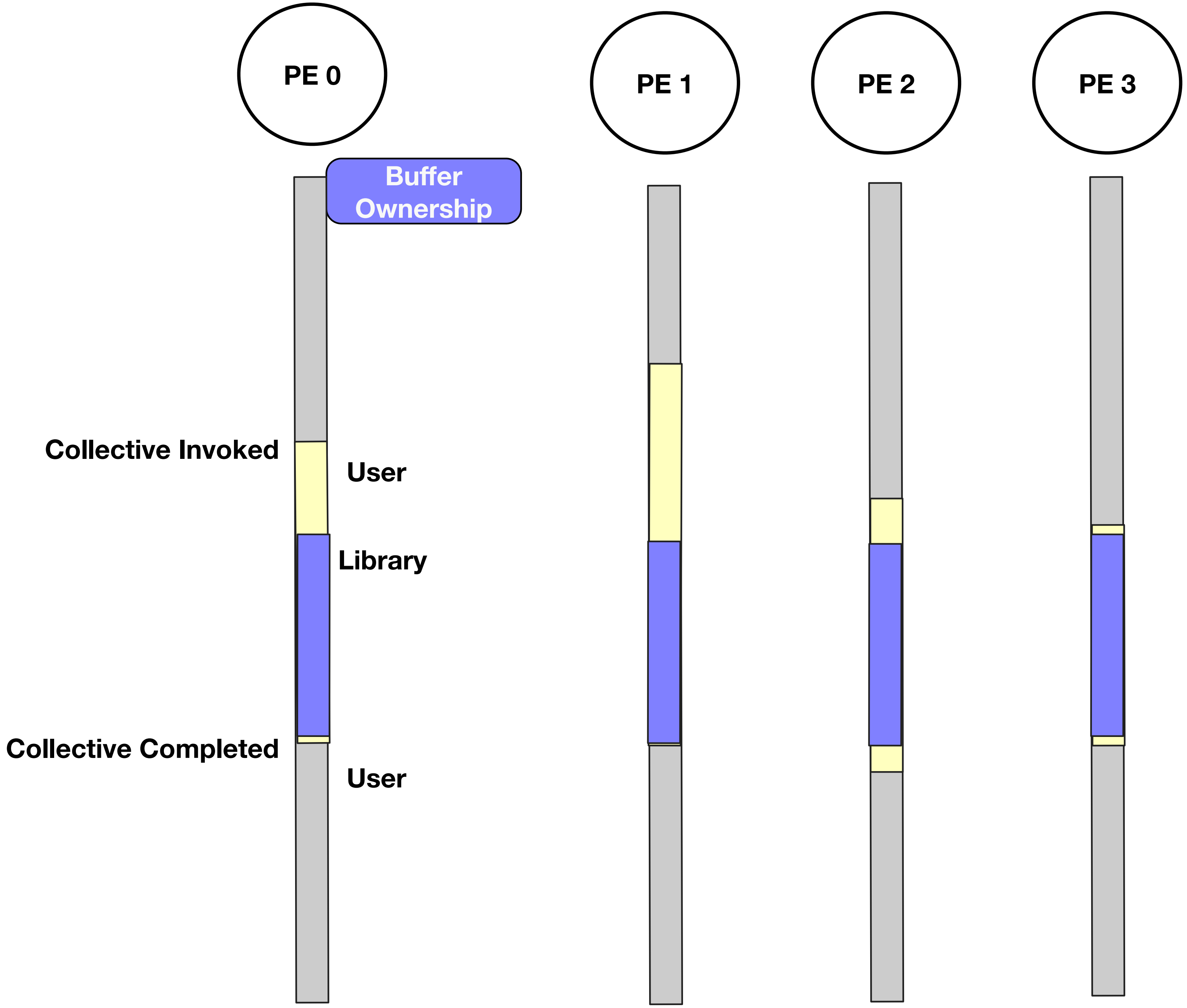
No Sync Collective Operations: Buffer Ownership is a Local Decision



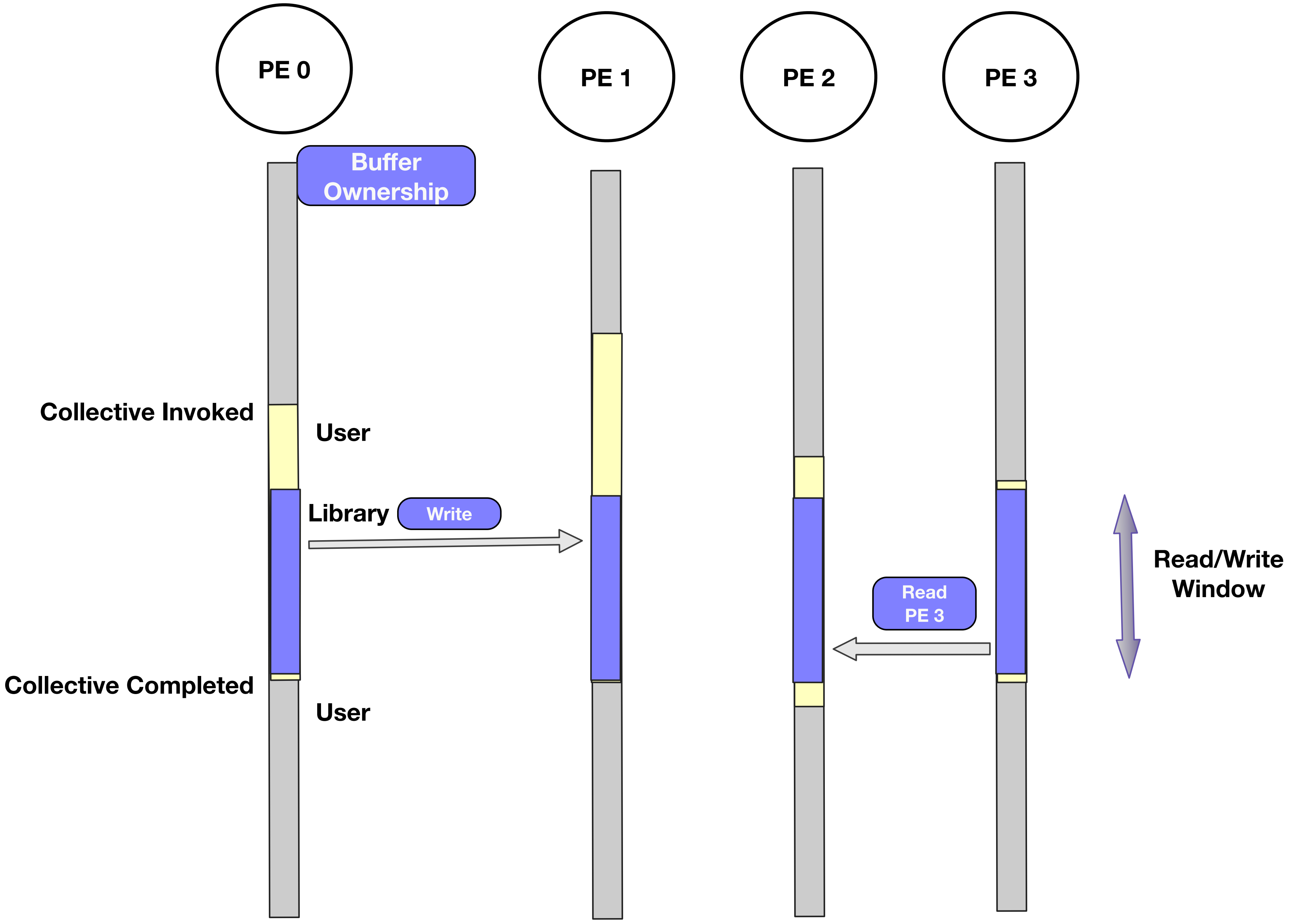
No Sync Collective Operations: Read and Write



Synchronized Collective Operations: Buffer Ownership



Synchronized Collective Operations: Read and Write



Creating Contexts : Design Choices

1. Local operation only
2. Collective operation only
3. Both local and collective
    1. Same interface
    2. Separate interfaces

Communication Context : Creating Context

Create operation as a collective operation

```
ucc_context_create(ucc_lib_t lib_obj, ucc_context_config_t ctx_config,
ucc_context_coll_oob_t *oob, ucc_context_t *comm_context);
```

Semantics:

- The main distinction between the interfaces is that this can be either a local or collective operation
  - When OOB is NULL, it is a local operation
  - When OOB collective is provided, it is a collective operation.
    - Resources cannot be decomposed into local and group resource
    - Resources need to be created in a group operation (Switch-based Collectives, Connection-based transports)

WG Feedback : Preference was for a single interface with both collective and local operation

- Move ucc_context_coll_oob_t to config
- Rename config to params throughout

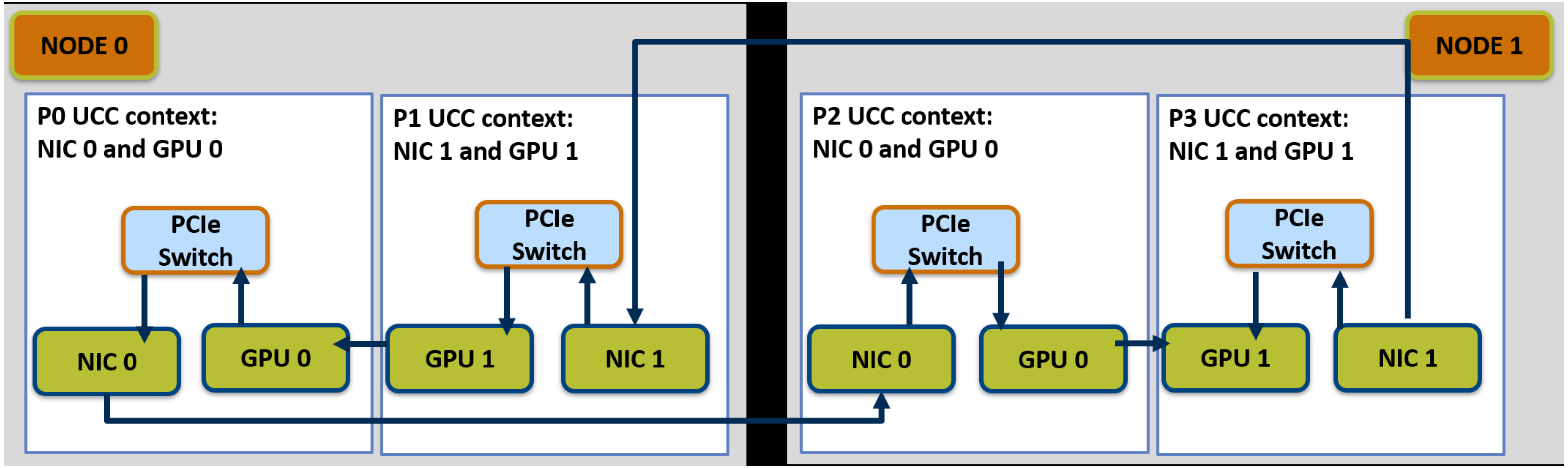
Device Abstraction and Affinity

- Device
  - Every context is coupled with a device
  - Device can be an HCA port, Memory, GPU Device, or combination of these devices

- How do you bind context to a device ?
  - Implicit model
    - Library decides the affinity of the resources created
  - Explicit model
    - The user explicitly requests affinity to a certain device (HCA port or device)

Explicit Model: Design and Usage

- The flow:
  - A process queries the UCC library for a list of supported devices (NICs - subset of those devices, need to derive abstract interface for that)
  - The process computes the distances from the GPU it is using and the NICs from the list. Finds the proper NIC based on distances.
  - The process modifies ucc_context_config data structure and specifies the selected NIC explicitly
  - A proper UCC context is created.



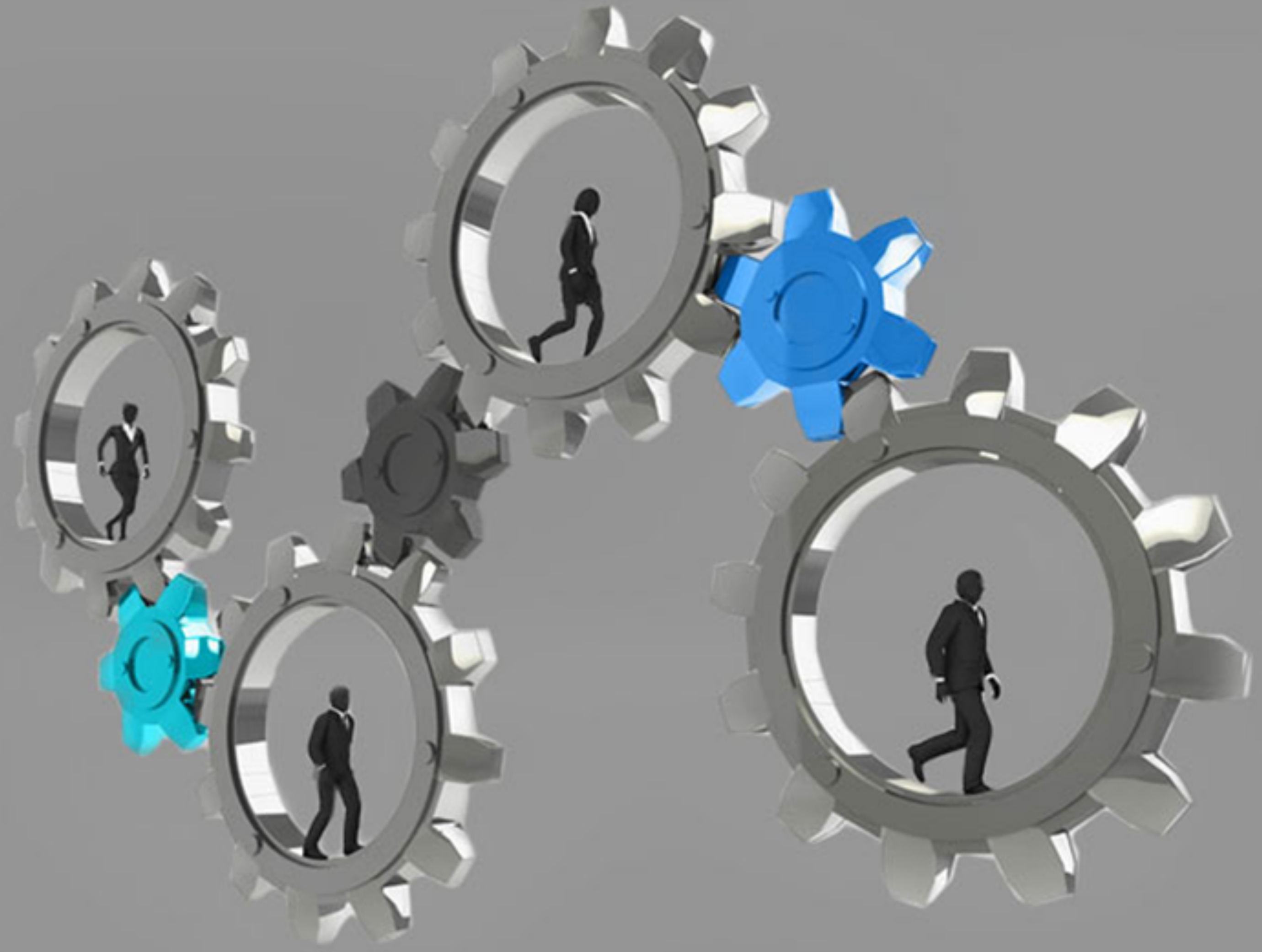
Open question : Explicit or Implicit model ?

- Explicit Model
  - Pros:
    - Fine-grained control for the user
    - Easier to support more use cases
  - Cons:
    - UCX does not provide interface for explicitly specifying the device
- Implicit Model
  - Pros:
    - The burden is on the library, not user
  - Cons:
    - Limited expressibility

WG Feedback : Explore explicit model and propose to the WG

ENABLER OF CO-DESIGN

UCF



Thank You

The UCF Consortium is a collaboration between industry, laboratories, and academia to create production grade communication frameworks and open standards for data centric and high-performance applications.

Abstractions

1. Collective Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operation
6. Task and task list
7. Plan

Team: Operations for creating teams

```
ucc_team_create_post(ucc_context_t  context, ucc_team_config_t comm_config,
oob_collectives_t oob_collectives, uint64_t *my_ep, ucc_team_t *new_team);


ucc_team_test(team);

ucc_team_destroy(team);
```

- 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

Team: Operations for creating teams

- Implementations should be ready to create invoke and execute after the team creation operation
  - 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
  - Exchange resource information
  - Agreement on the context configurations
  - Agreement on the endpoints

Team : Customizing team

```
struct ucc_team_config_t {
     ucc_team_config_mask_t mask;
    ucc_post_ordering ordering;
    uint64_t num_outstanding_collectives;
    ucc_collective_sync_type_t sync;
    ucc_ep_range_contig ep_range;
    ucc_ep_flag in_out;
   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
- 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

Team : Query Operations

An object to encapsulate local resource and express network parallelism

```
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);
```

Semantics:

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

Team : Splitting 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

Abstractions

1. Collective Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operation
6. Task and task list
7. Plan

Endpoint

- Endpoint is an address for communication. It can be bound to the thread or process.
  - Provides a way to address the UCC context (resources)
  - Provides a globally addressable name for the contexts

Semantics

- A set of endpoints form the team
- The endpoint is an integer (uint64_t) representing the resource
  - It can be provided as input (typically mapped from the programming model)
  - It can be provided as output from team creation operation

Endpoints as input and Output

- Endpoint as an IN parameter
  - User can pass rank/openshmem index as an endpoint.
  - Ordering is established by the User
  - User provides a hint about the endpoint range, whether it is ordered or not. This will provide a hint to optimize for the user
  - Library maintains the mapping between endpoint indexes and internal endpoints (UCP endpoints, hardware indexes)

- Endpoint as an OUT parameter
  - The library will create a list of endpoints.
  - The ordering of the endpoints is established by the library
  - The library provides interfaces for the list of endpoints, my endpoint, and translation
  - The User manages the mapping between the ranks and endpoints by doing an all gather above UCC

Endpoints: Team create operations

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

- Team creation only with a collective operation on the newly created team
- Support spawn semantics .i.e., supports adding an endpoint to the team

Endpoint based implementation is not explored yet in XCCL

Endpoints: Team create operations

ucc_create_team_from_ep_list(ucc_team_t parent_ucc_team, uint64_t *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, uint64_t ep, ucc_team_t *new_team);

Open questions:

- Should team created by endpoints be a local operation ?
- Light-weight team creation by passing the list of endpoints
    - Enables lazy resource allocation
- Should team created by endpoints be of different type ?

Abstractions

1. Collective Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operation
6. Task and task list
7. Plan

Collective Operations : Building blocks (1)

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


int ucc_collective_post(ucc_coll_req request)
int ucc_collective_test(ucc_coll_req request);
int ucc_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
- Finalize
  - Releases the resources for the collective operation represented by the request
  - The post and wait operations are invalid after finalize

Collective Operations : Building blocks (3)

```
ucc_collective_init( ucc_coll_op_args *coll_args, ucc_team_t team, ucc_coll_req *coll_req);

ucc_collective_init_and_post( ucc_coll_op_args *coll_args, ucc_team_t team, ucc_coll_req *request);

int ucc_collective_post(ucc_coll_req request)

int ucc_collective_test(ucc_coll_req request);

int ucc_collective_finalize(ucc_coll_req request);
```

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

Customizing Collective Operation (1)

```
typedef struct ucc_collective_op_arguments
{
        ucc_collop_config_mask_t mask;
        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;
        uint64_t root_ep;
} ucc_coll_op_args_t;
```

- 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 : Operations and Reductions (2)

```
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

```
struct ucc_coll_buffer_info_t {
        ucc_collbuf_config_mask_t mask;
        void *src_buffer;
        uint32_t *scounts;
        uint32_t *src_displacements;
        void *dst_buffer;
        uint32_t *dst_counts;
        uint32_t *dst_displacements;
        size_t size;
        int64 flags; /* in-buffer */
        ucc_dt_type_t src_datatype;
        ucc_dt_type_t dst_datatype;
}
```

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

Customizing Collective Operation (4): 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

Abstractions

1. Collective Library
2. Communication Context
3. Teams
4. Endpoints
5. Collective Operation
6. Task and task list
7. Plan

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
- Combining computation + collective operation
- Bundled collective operations

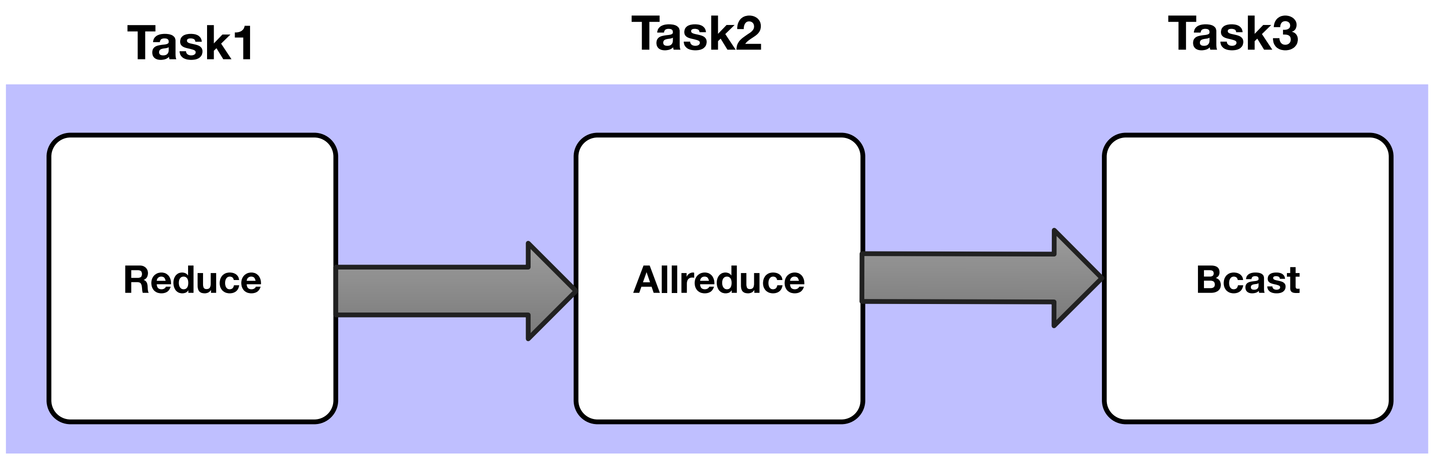
How to express groups of collectives?

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

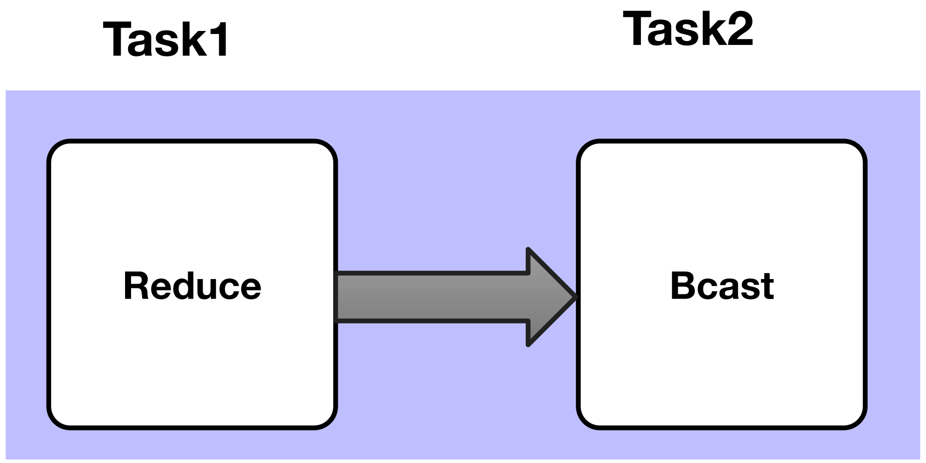
Collective Groups: Task and Task List

- 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



Task list for Allreduce (leader process)



Task list for Allreduce (non-leader process)

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_list);
```

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

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);

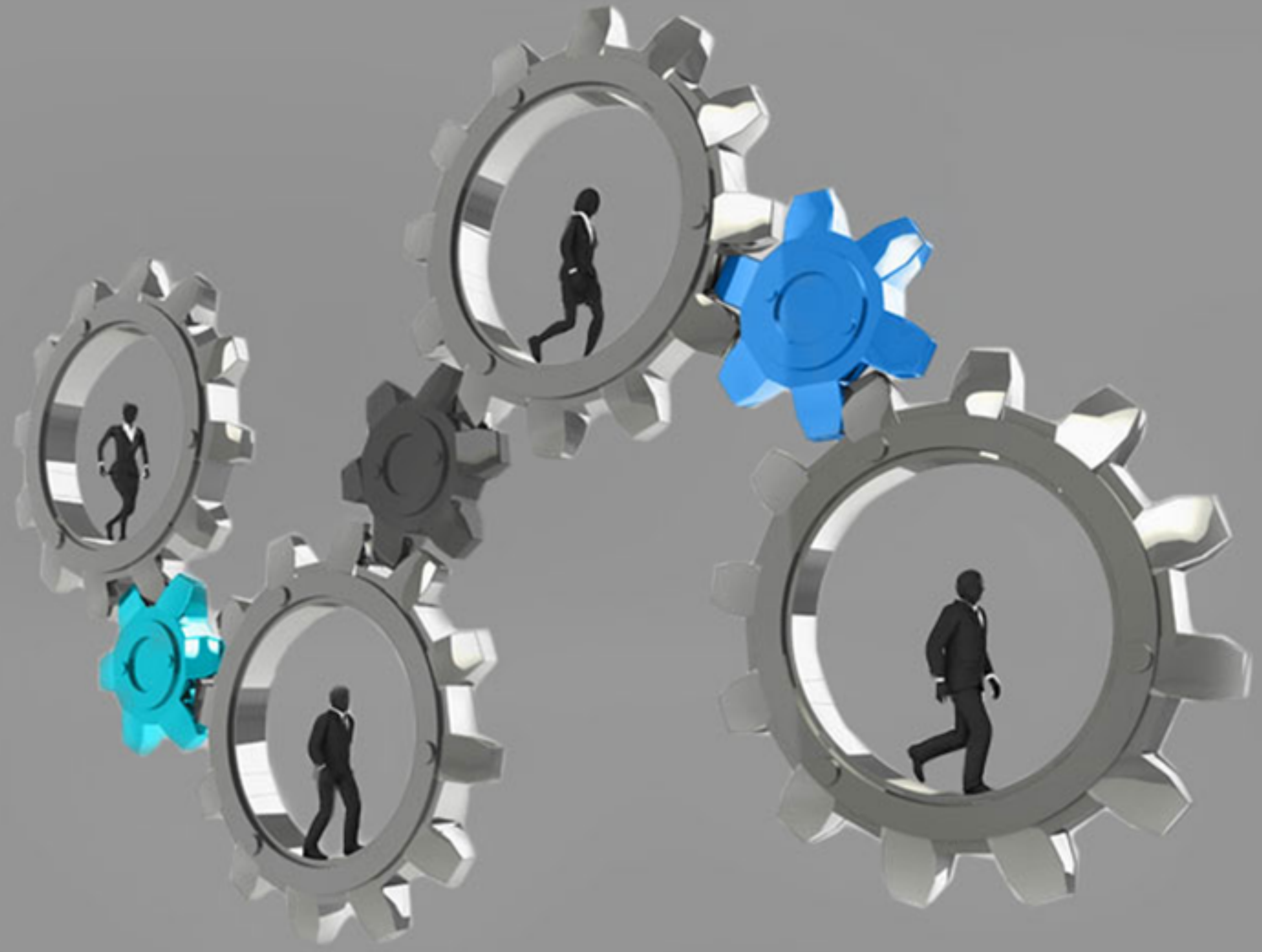
Semantics:

- 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

ENABLER OF CO-DESIGN



Thank You

The UCF Consortium is a collaboration between industry, laboratories, and academia to create production grade communication frameworks and open standards for data centric and high-performance applications.