UCC Overview

Manjunath Gorentla Venkata, Valentin Petrov, Sergey Lebedev, Ferrol Aderholdt

On behalf of UCC WG @ UCF Annual Event, 2021

Unified Collective Communication

Open-source project to provide an API and library implementation of collective (group) communication operations

Outline

Manju

- UCC Design, API and Roadmap

Valentin

- UCC Hierarchical Collectives

Sergey

- UCC GPU Collectives

Ferrol

- UCC RMA/One-sided Collectives

UCC Design Challenges

- Unified collective stack for HPC and DL/ML workloads
  - Need to support a wide variety of semantics
  - Need to optimize for different performance sensitives - latency, bandwidth, throughput
  - Need for flexible resource scheduling and ordering model
- Unified collective stack for software and hardware transports
  - Need for complex resource management - scheduling, sharing, and exhaustion
  - Need to support multiple semantic differences – reliability, completion
- Unify parallelism and concurrency
  - Concurrency – progress of a collective and the computation
  - Parallelism – progress of many independent collectives
- Unify execution models for CPU, GPU, and DPU collectives
  - Two-way execution model – control operations are tightly integrated
    - Do active progress, returns values, errors, and callbacks with less overhead
  - One-way execution model – control operations are loosely integrated
    - passive progress, and handle return values (GPU/DPUs)

UCC Design Principles: Properties desired in the solution

- Scalability and performance for key use-cases
  - Enable efficient implementation for common cases in MPI, OpenSHMEM and AI/ML
- Extensible
  - We cannot possibly cover all the options and features for all use cases
  - We need the API and semantics that is modular
- Opt in-and-out
  - If for a certain path some semantic is not applicable, we need a way to opt-out
- Explicit API and semantics over implicit
  - Explicit -> implicit is easier than implicit -> explicit
- Minimal API surface area
  - Lessen the mental load
  - A few set of abstractions to understand and go into details when required
- Other properties are such as the ability to override functionality, programmability, expressing general and specific functionality are important

UCC’s Solution: Three important concepts

- Abstractions
  - Abstract the resources required for collective operations
  - Local: Library, Context, Endpoints, Execution Engine
  - Global: Teams
- Operations
  - Defines how to interact with the abstractions
  - Create/modify/destroy the resources
  - Build, launch and finalize collectives
- Properties
  - Defines how to customize abstractions and operations
  - Explicit way to request for optional features, semantics, and optimizations (opt-in or opt-out model)
  - Provides an ability to express and request many cross-cutting features
  - Properties are preferences expressed by the user of the library and what the library provides is queried
- Details of concepts
  - Code: https://github.com/openucx/ucc
  - Slides: https://github.com/manjugv/ucc_wg_public

Concepts

- Abstractions for Resources
  - Collective Library
  - Communication Context
  - Teams
- Collective Operations
- Triggered Operations

UCC Library

- An object to encapsulate resources related to the group communication operations

Semantics

- All UCC operations should be invoked between the init and finalize operations.
- The library can be tailored to match the user requirements
- The user of the library can be parallel programming models (MPI, PGAS/OpenSHMEM, PyTorch) or applications

Operations

- Routines for initializing and finalizing the resources for the library.

Library Init C Interface

```c
/**
 *  @ingroup UCC_LIB
 *
 *  @brief The @ref ucc_init initializes the UCC library.
 *
 *  @param [in]  params    user provided parameters to customize the library functionality
 *  @param [in]  config    UCC configuration descriptor allocated through
 *                         @ref ucc_lib_config_read "ucc_config_read()" routine.
 *  @param [out] lib_p     UCC library handle
 *
 *  @parblock
 *
 *  @b Description
 *
 *  A local operation to initialize and allocate the resources for the UCC
 *  operations. The parameters passed using the ucc_lib_params_t and
 *  @ref ucc_lib_config_h structures will customize and select the functionality of the
 *  UCC library. The library can be customized for its interaction with the user
 *  threads, types of collective operations, and reductions supported.
 *  On success, the library object will be created and ucc_status_t will return
 *  UCC_OK. On error, the library object will not be created and corresponding
 *  error code as defined by @ref ucc_status_t is returned.
 *
 *  @endparblock
 *
 *  @return Error code as defined by @ref ucc_status_t
 */

static inline ucc_status_t ucc_init(const ucc_lib_params_t *params,
                                    const ucc_lib_config_h config,
                                    ucc_lib_h *lib_p)
{
    return ucc_init_version(UCC_API_MAJOR, UCC_API_MINOR, params, config,
                            lib_p);
}
```

Properties: Collectives LIBRARY

```
/**
 *
 *  @ingroup UCC_LIB_INIT_DT
 *
 *  @brief Structure representing the parameters to customize the library
 *
 *  @parblock
 *
 *  Description
 *
 *  @ref ucc_lib_params_t defines the parameters that can be used to customize
 *  the library. The bits in "mask" bit array is defined by @ref
 *  ucc_lib_params_field, which correspond to fields in structure @ref
 *  ucc_lib_params_t. The valid fields of the structure is specified by the
 *  setting the bit to "1" in the bit-array "mask". When bits corresponding to
 *  the fields is not set, the fields are not defined.
 *
 *  @endparblock
 *
 */
typedef struct ucc_lib_params {
    uint64_t                mask;
    ucc_thread_mode_t       thread_mode;
    uint64_t                coll_types;
    uint64_t                reduction_types;
    ucc_coll_sync_type_t    sync_type;
    ucc_reduction_wrapper_t reduction_wrapper;
} ucc_lib_params_t;
```

- Thread Model
- Collective Types
- Reduction Types
- Synchronization Types

Properties of library: Thread model

- UCC_LIB_THREAD_SINGLE:
  - The user program cannot be multithreaded
- UCC_LIB_THREAD_FUNNELED:
  - The user program may be multithreaded, however, only one thread should invoke the UCC interfaces
- UCC_LIB_THREAD_MULTIPLE:
  - The user program can be multithreaded, and any any thread may invoke the UCC operations.

Concepts

- Abstractions for Resources
  - Collective Library
  - Communication Context
  - Teams
- Collective Operations
- Triggered Operations

Communication Context (1)

- An object to encapsulate local resource and express network parallelism

- Context is created by ucc_context_create()
- 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
- Context can be bound to a specific core, socket, or an accelerator
  - Provides an ability to express affinity
- Context can be used to control resource sharing
- Multiple contexts per team (from same thread) can be supported
  - Software and hardware collectives
  - Optimize for bandwidth utilization

Context Create C Interface

```c
/**
 *  @ingroup UCC_CONTEXT
 *
 *  @brief The @ref ucc_context_create routine creates the context handle.
 *
 *  @param [in]   lib_handle  Library handle
 *  @param [in]   params      Customizations for the communication context
 *  @param [in]   config      Configuration for the communication context to read
 *                            from environment
 *  @param [out]  context     Pointer to the newly created communication context
 *
 *  @parblock
 *
 *  @b Description
 *
 *  The @ref ucc_context_create creates the context and @ref ucc_context_destroy
 *  releases the resources and destroys the context state. The creation of
 *  context does not necessarily indicate its readiness to be used for
 *  collective or other group operations. On success, the context handle will be
 *  created and ucc_status_t will return UCC_OK. On error, the library object
 *  will not be created and corresponding error code as defined by
 *  @ref ucc_status_t is returned.
 *
 *  @endparblock
 *
 *  @return Error code as defined by @ref ucc_status_t
 */


ucc_status_t ucc_context_create(ucc_lib_h lib_handle,
                                const ucc_context_params_t *params,
                                const ucc_context_config_h  config,
                                ucc_context_h *context);
```

PROPERTIES OF 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

Concepts

- Abstractions for Resources
  - Collective Library
  - Communication Context
  - Teams
- Collective Operations
- Triggered Operations

Teams

- An object to encapsulate the resources required for group operations such as collective communication operations.
- 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 Create Interface

```
/**
 *  @ingroup UCC_TEAM
 *
 *  @brief The routine is a method to create the team.
 *
 *  @param
  [in]  contexts           Communication contexts abstracting the resources
 *  @param
  [in]  num_contexts       Number of contexts passed for the create operation
 *  @param  [in]  team_params        User defined configurations for the team
 *  @param  [out] new_team           Team handle
 *
 *  @parblock
 *
 *  @b Description
 *
 *  @ref ucc_team_create_post is a nonblocking collective operation to create
 *  the team handle. It takes in parameters ucc_context_h and ucc_team_params_t.
 *  The ucc_team_params_t provides user configuration to customize the team and,
 *  ucc_context_h provides the resources for the team and collectives.
 *  The routine returns immediately after posting the operation with the
 *  new team handle. However, the team handle is not ready for posting
 *  the collective operation. ucc_team_create_test operation is used to learn
 *  the status of the new team handle. On error, the team handle will not
 *  be created and corresponding error code as defined by @ref ucc_status_t is
 *  returned.
 *
 *  @endparblock
 *
 *  @return Error code as defined by @ref ucc_status_t
 */
ucc_status_t ucc_team_create_post(ucc_context_h *contexts,
                                  uint32_t num_contexts,
                                  const ucc_team_params_t *team_params,
                                  ucc_team_h *new_team);
```

PROPERTIES: Teams

- 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 be in a contiguous range ?
- Synchronization Model
  - On_Entry, On_Exit, or On_Both – this helps with global resource allocation
- Datatype
  - Can be customized for contiguous, strided, or non-contiguous datatypes

```c
typedef struct ucc_team_params {
    uint64_t                 mask;
    ucc_post_ordering_t      ordering;
    uint64_t                 outstanding_colls;
    uint64_t                 ep;
    uint64_t                 *ep_list;
    ucc_ep_range_type_t      ep_range;
    uint64_t                 team_size;
    ucc_coll_sync_type_t     sync_type;
    ucc_team_oob_coll_t      oob;
    ucc_team_p2p_conn_t      p2p_conn;
    ucc_mem_map_params_t     mem_params;
    ucc_ep_map_t             ep_map;
    uint64_t                 id;
} ucc_team_params_t;
```

Concepts

- Abstractions for Resources
  - Collective Library
  - Communication Context
  - Teams
- Collective Operations
- Triggered Operations

Collective Operations: Building Blocks

```
ucc_status_t ucc_collective_init(ucc_coll_op_args_t* coll_args,
                                 ucc_coll_req_h* request, ucc_team_h team);

ucc_status_t ucc_collective_post(ucc_coll_req_h request);

ucc_status_t ucc_collective_init_and_post(ucc_coll_op_args_t* coll_args,
                                          ucc_coll_req_h* request,
                                          ucc_team_h team);

ucc_status_t ucc_collective_finalize(ucc_coll_req_h request);
```

Collective Operations: BUILDING BLOCKs (2)

Semantics

- Collective operations : ucc_collective_init(…) and ucc_collective_init_and_post(…)
- Local operations: ucc_collective_post, test, and 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)

- 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

Concepts

- Abstractions for Resources
  - Collective Library
  - Communication Context
  - Teams
- Collective Operations
- Triggered Operations

UCC Execution Engine, Events, and Triggered Operations

Execution Engine

- It is an execution context that supports event-driven network execution on the CUDA streams, CPU threads, and DPU threads.

Events

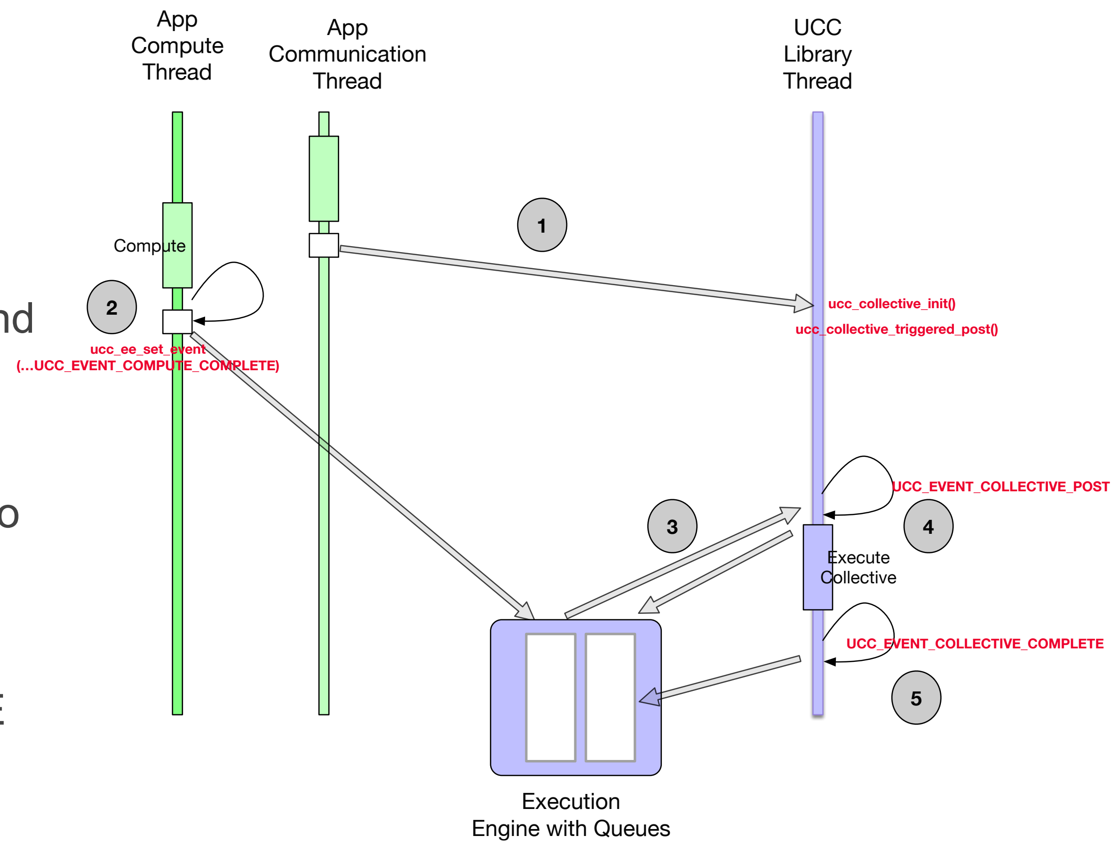
- Library-generated events
  - Examples: Completion of operation, launch of collective
- User-generated events
  - Examples: Compute complete, Data-ready

Triggered Operations

- Triggered operations enable the posting of operations on an event.
  - UCC supports triggering collective operations by library-generated and user-generated events.
- Team-level customization to enable/disable triggered operations

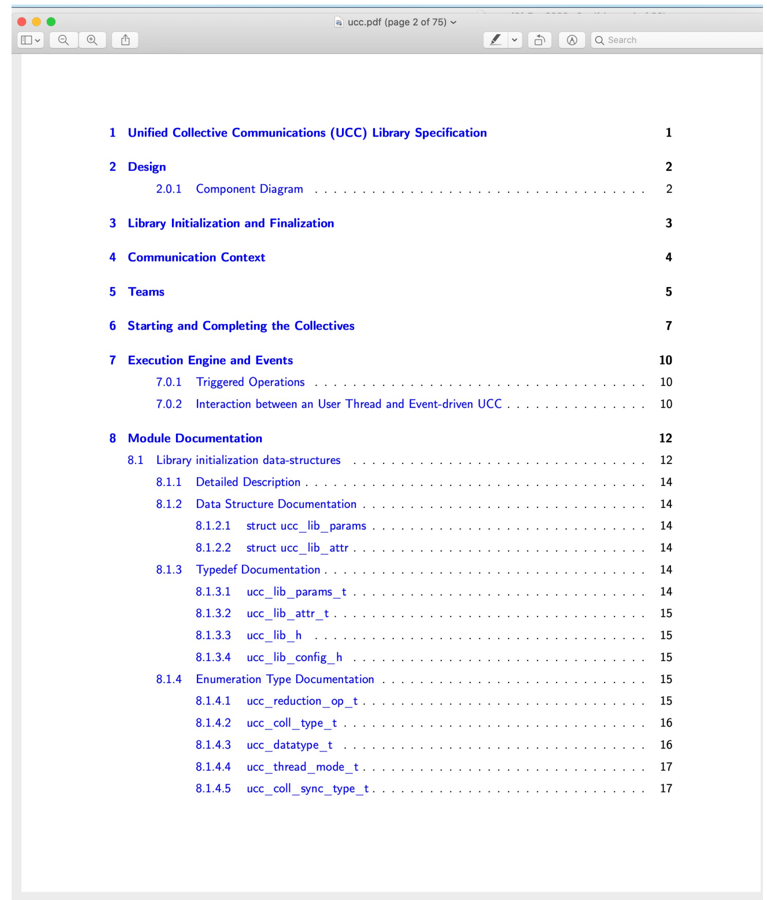
UCC Events: Interaction between a User Thread and Event-driven UCC

1. Application initializes the collective operation
2. When the application completes the compute, it posts the UCC_EVENT_COMPUTE_COMPLETE event to the execution engine.
3. The library thread polls the event queue and triggers the operations that are related to the compute event.
4. The library posts the UCC_EVENT_POST_COMPLETE event to the event queue.
5. On completion of the collective operation, the library posts UCC_EVENT_COLLECTIVE_COMPLETE event to the completion event queue.

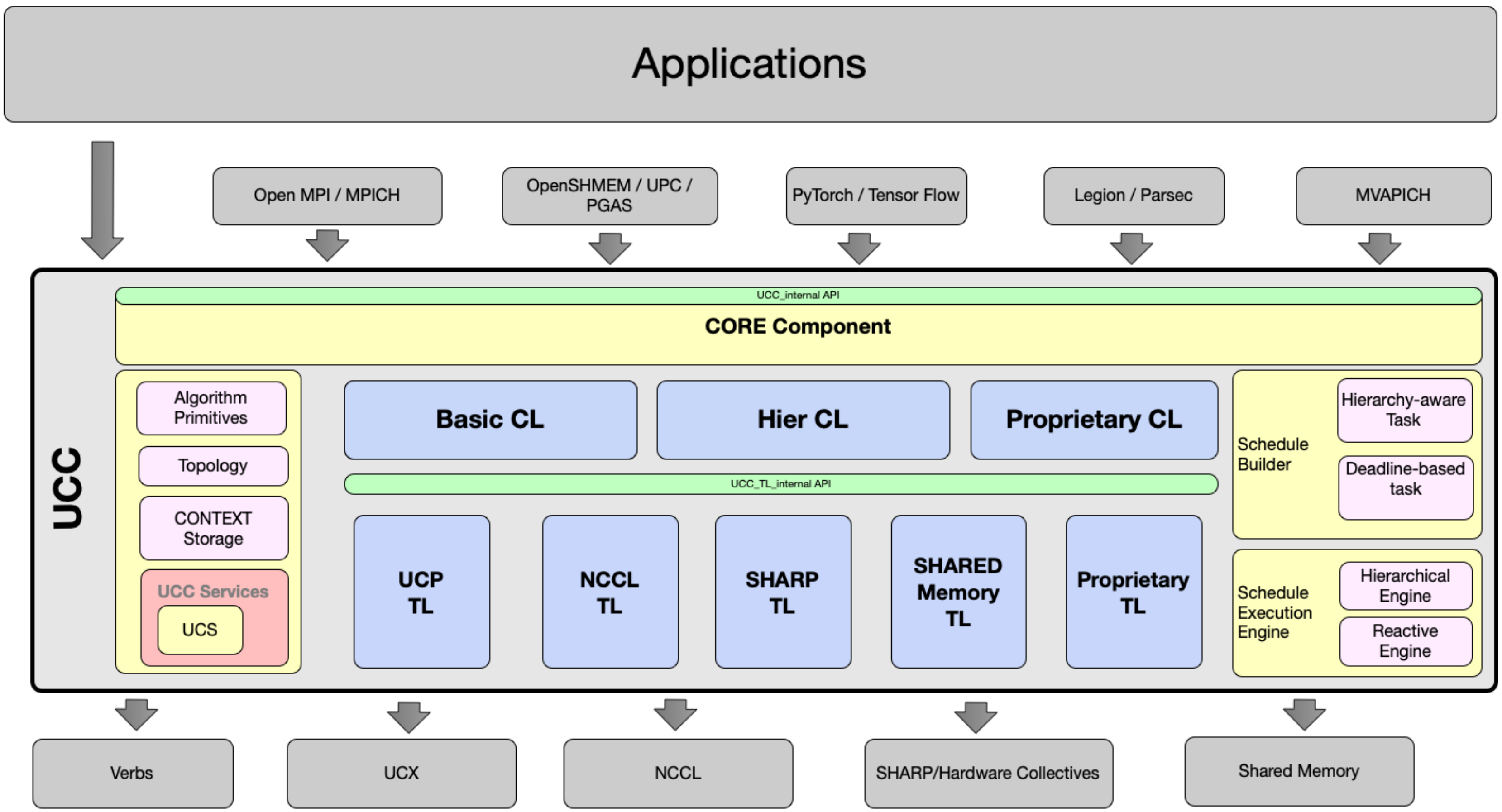


UCC Specification: Interfaces and semantics fully specified

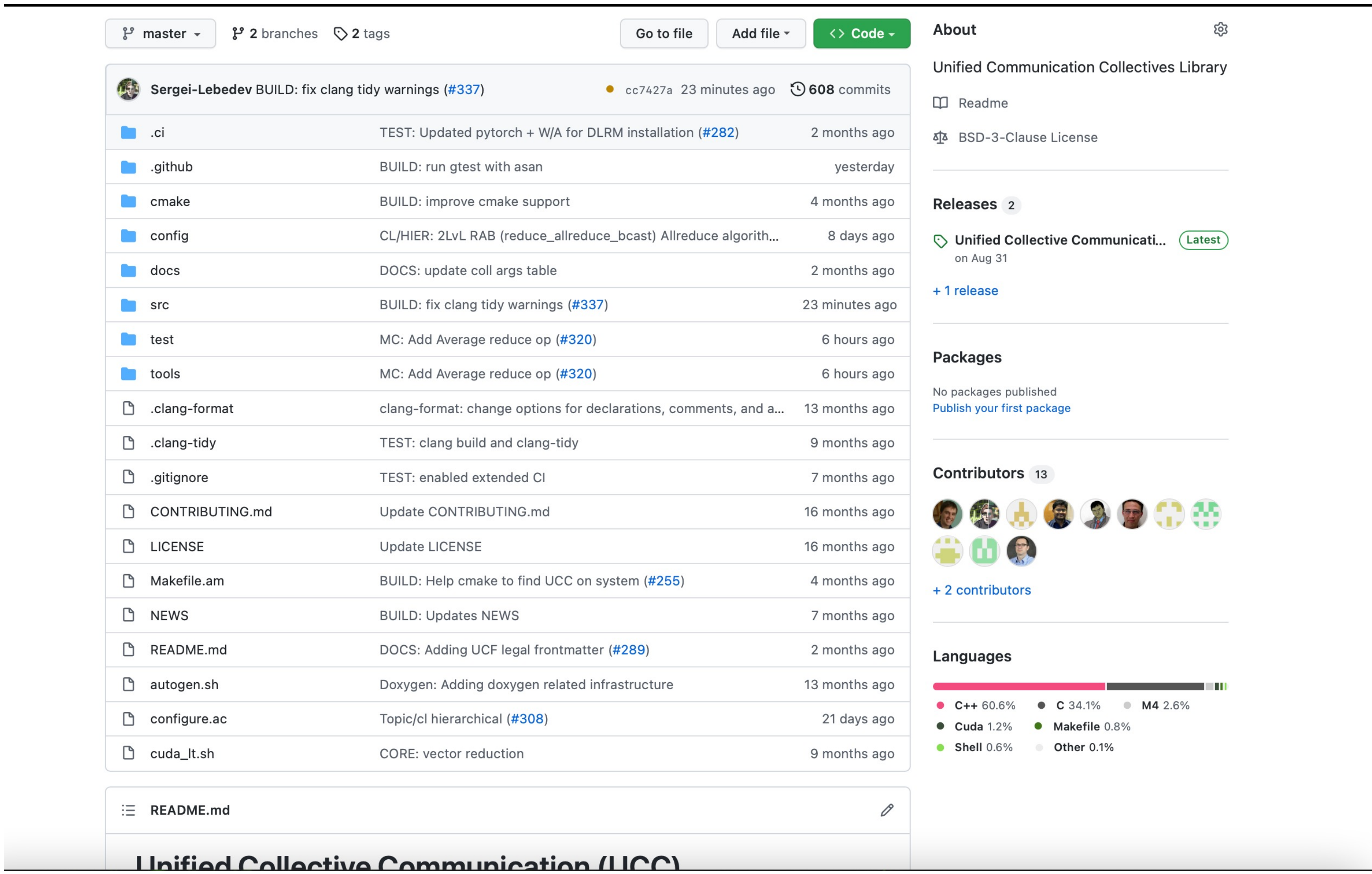
- Specification available on the UCC GH
- Specification is ahead of the code now
- The version 1.0 is agreed by the working group and merged into the master branch
- Over 100 pages of detailed information about the interfaces and semantics
- Doxygen based documentation
- Both pdf and html available


ucc.pdf (page 2 of 75)

UCC Reference Implementation: Component Diagram



UCC: Reference Implementation Status



Integration with HPC and DL Programming Models

- Open MPI
  - Available in Open MPI v4.1 and above
  - Support alltoall, broadcast, barrier, allreduce, and allgather collective operations
- Open MPI/OSHMEM
  - Supports OpenSHMEM v1.4 collective operations
  - Also enables collective operations based on one-sided RMA operations
- PyTorch
  - Support via third-party plugin : https://github.com/facebookresearch/torch_ucc

UCC v1.0 Expected to Release Q4 2021

- v0.1.0 Early Release (Released Aug 31st, 2021)
  - Support for most collectives required by parallel programming models
  - Many algorithms to support various data sizes, types, and system configurations
  - Support for CPU and GPU collectives
  - Collectives on basic datatypes
  - Testing infrastructure
    - Unit tests, profiling, and performance tests
  - Support for MPI and PyTorch (via Third-party plugin)

- v1.0 Major Release (Expected 2021)
  - Hardware collectives - support for SHARP
  - Support for more optimized collectives (hierarchical)
  - Support for OpenSHMEM with one-sided collectives and active sets
  - Support for NCCL collectives
  - Infrastructure for pipelining, task management , and customization (algorithm selection)
  - Collectives on generic datatypes (in discussion)
  - Incorporate feedback from v0.1.0 release

Contributions are Welcome!

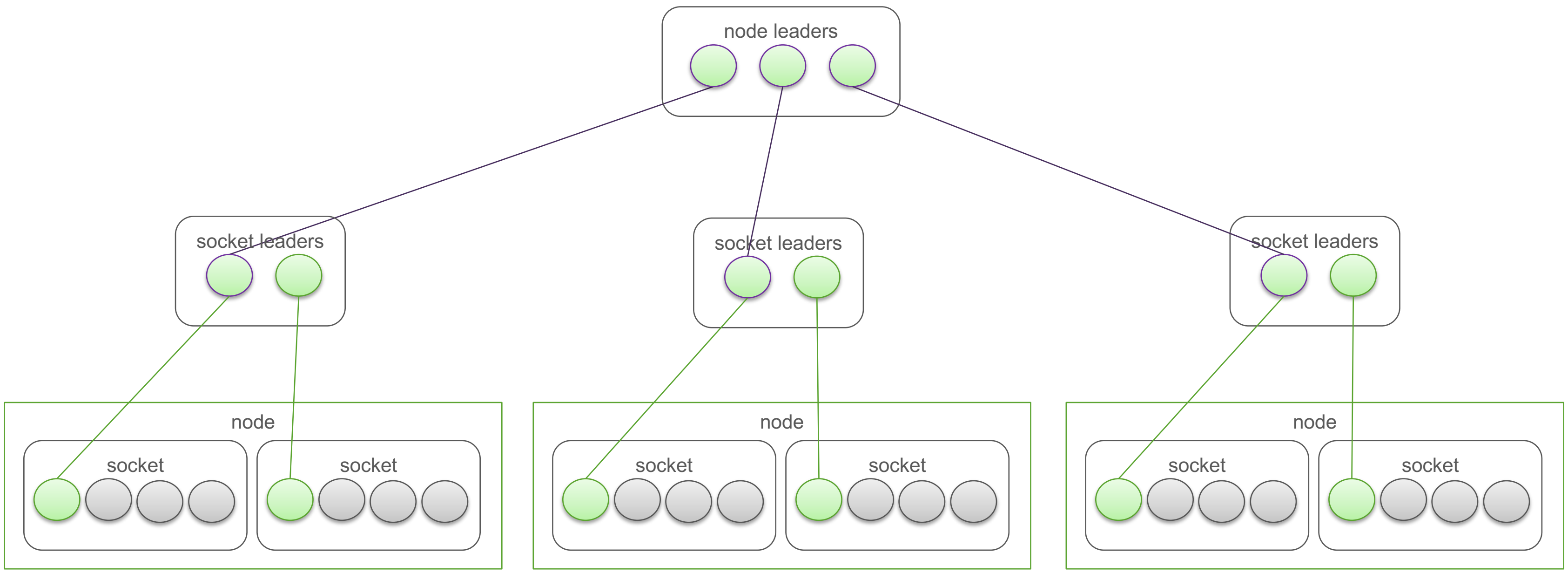
- What contributions are welcomed ?
  - Everything from design, documentation, code, testing infrastructure, code reviews …
- How to participate ?
  - WG Meetings : https://github.com/openucx/ucc/wiki/UCF-Collectives-Working-Group
  - GitHUB: https://github.com/openucx/ucc
  - Slack channel: Ask for an invite
  - Mailing list: ucx-group@elist.ornl.gov

UCC Hierarchal Collectives

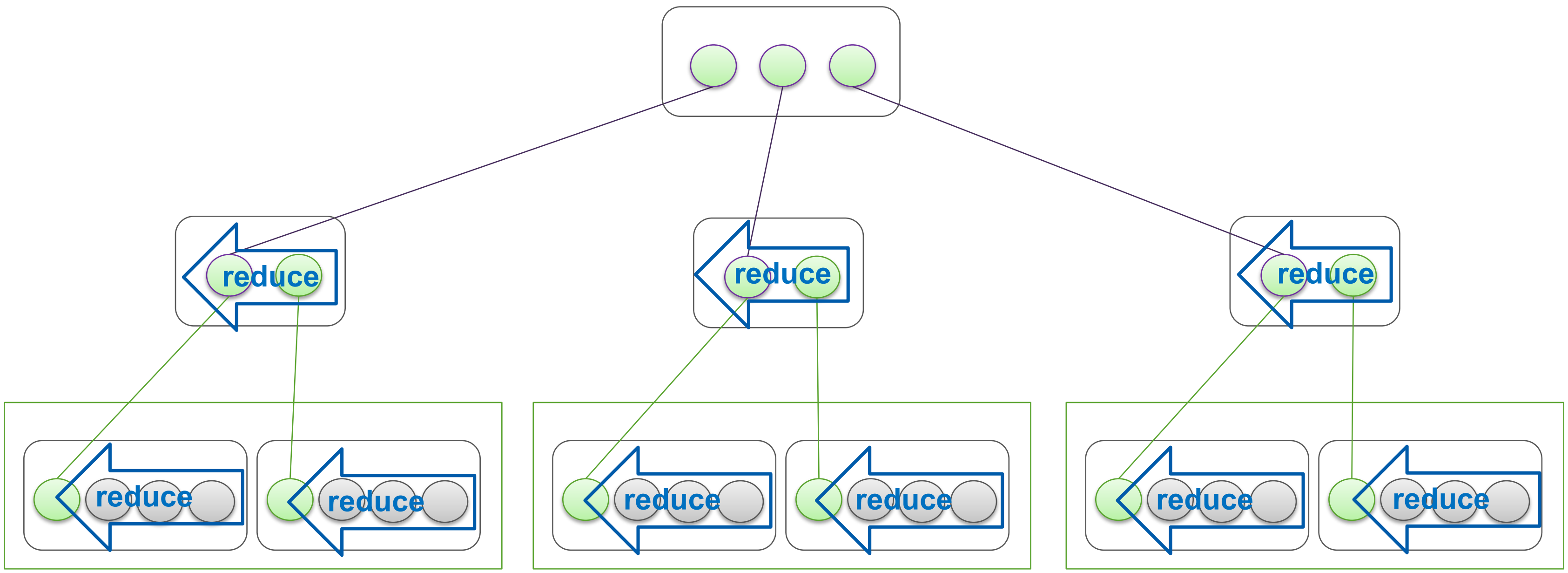
What is Hierarchical collective

- UCC Team (communicator) is split into subgroups of processes that can form a topological hierarchy
- A collective operation over original UCC team can be then implemented as a combination of smaller collectives over subgroups
- Technique is commonly used: ompi/cheetah, hcoll
- Can be beneficial for latency and bw bound collectives depending on configuration

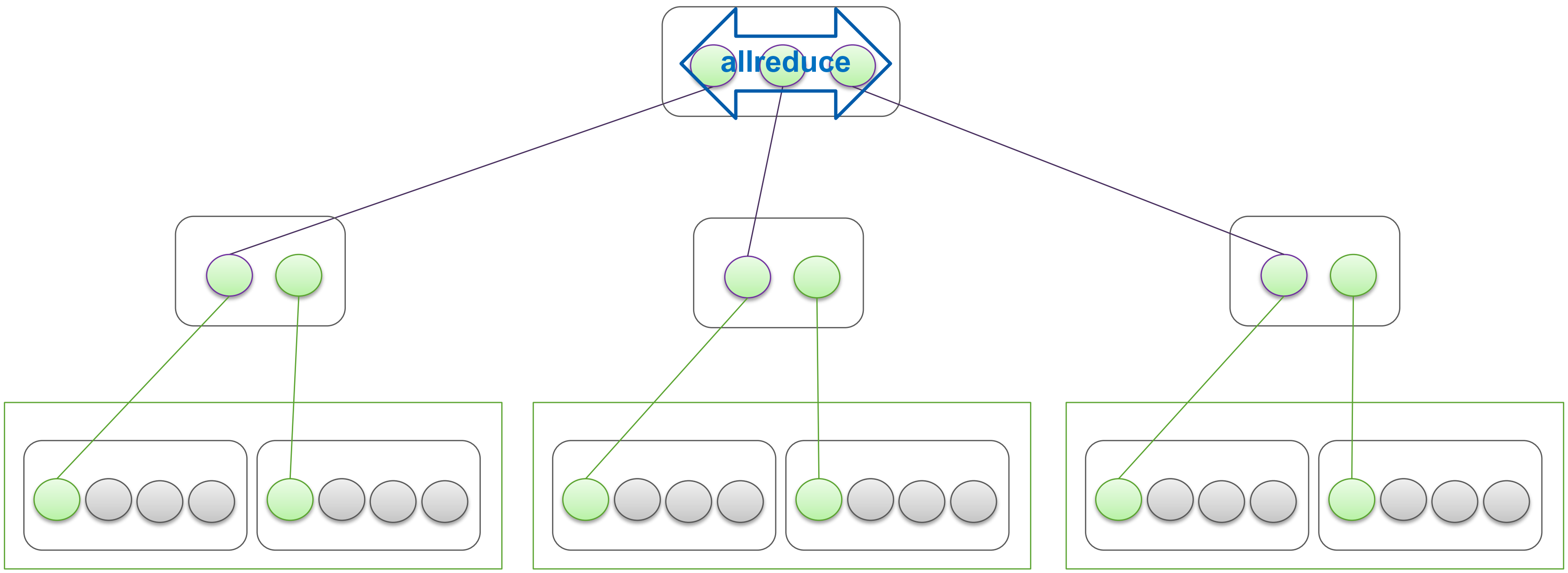
Example of subgrouping



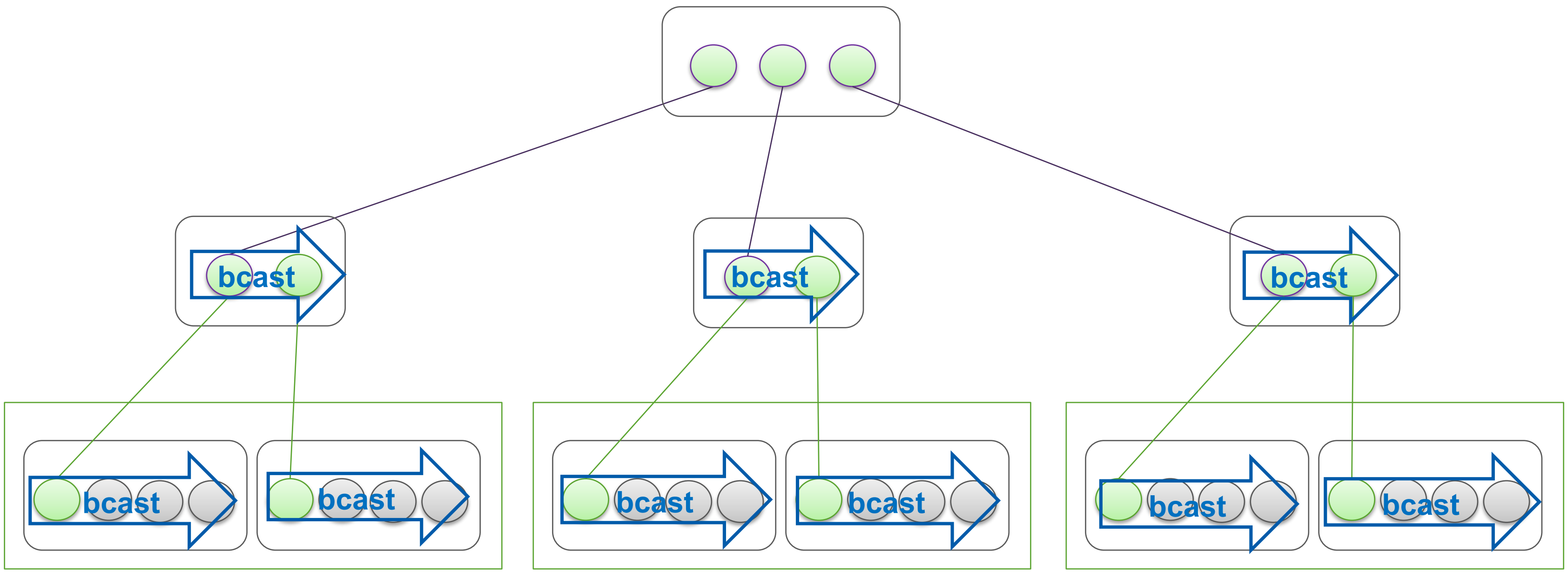
Example of hierarchical Allreduce



Example of hierarchical Allreduce

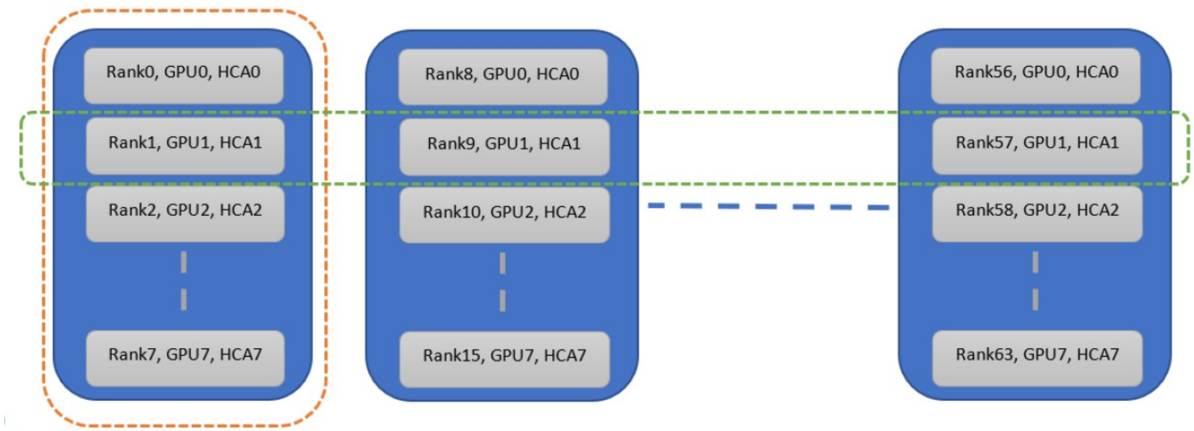


Example of hierarchical Allreduce



Example Allreduce, 2lvl

- 2 level hierarchical allreduce applicable to communicators with constant ppn (number of ranks per node is the same on all nodes)
- The ranks of communicator are split into groups: NODE_GROUP and NET_GROUP. Each rank is part of 2 groups.
  - NODE_GROUP contains all the ranks of the comm that belong to the same node (host); each rank is assigned a local NODE_RANK (relative to the node group). Size of the NODE_GROUP = ppn
  - NET GROUP contains the ranks from all the nodes of the communicator with the same NODE_RANK. Size of the NET_GROUP = nnodes.
  - There are total "nnodes" NODE_GROUPS and "ppn" "NET_GROUPS"
- The algorithm on each process consists of 3 steps: 1. ReduceScatter over NODE_GROUP, 2. Allreduce over NET_GROUP, 3. Allgather over NODE_GROUP.



UCC: subgrouping

- src/topo framework
- Local process_info (bound socket/numa id, pid, hosthash, etc) is imbedded into ucc_context_address
- Addr exchange during either ucc_context_create or ucc_team_create
- Topo structure: sorted array of proc_info
- Subgroups discovery: purely local procedure
- Topo initialization is performed “on demand” – when TL/CL reports that “topo” is required
- CL/HIER – CL responsible for implementation of hierarchical collectives
  - Splits the CORE UCC team into subgroups
  - Initializes required TL teams per subgroup and constructs score_maps
  - Builds collective schedules (ie hierarchical algorithms)

Schedules & Tasks

- ucc_coll_task_t – is a quantum of work at TL level
  - Describes a collective operation over group of processes.

```
typedef ucc_status_t
(*ucc_base_coll_init_fn_t)(ucc_base_coll_args_t *coll_args,
                           ucc_base_team_t *team,
                           ucc_coll_task_t **task);


typedef struct ucc_coll_task {
    ucc_coll_req_t super;
    uint32_t flags;
    ucc_base_coll_args_t bargs; ucc_base_team_t *team; //CL/TL team pointer
    ucc_coll_post_fn_t post;
    ucc_coll_triggered_post_fn_t triggered_post;
    ucc_coll_finalize_fn_t finalize; ucc_coll_callback_t cb;
    ucc_event_manager_t em;
    ucc_status_t (*progress)(struct ucc_coll_task *self);
    struct ucc_schedule *schedule;
    ucc_ee_h ee;
    ucc_ev_t *ev;
    void *ee_task;
    ucc_coll_task_t *triggered_task;
    union { /* used for st & locked mt progress queue */
        ucc_list_link_t list_elem; /* used for lf mt progress queue */
        ucc_lf_queue_elem_t lf_elem;
    };
    uint8_t n_deps;
    uint8_t n_deps_satisfied;
    uint8_t n_deps_base;
    double start_time; /* timestamp of the start time: either post or triggered_post
*/
    uint32_t seq_num;
} ucc_coll_task_t;
```

- ucc_schedule_t : collection of tasks connected via event_manager

```
typedef enum {
    UCC_EVENT_COMPLETED = 0,
    UCC_EVENT_SCHEDULE_STARTED,
    UCC_EVENT_TASK_STARTED,
    UCC_EVENT_ERROR,
    UCC_EVENT_LAST
} ucc_event_t;

typedef struct ucc_schedule {
    ucc_coll_task_t super;
    int n_completed_tasks;
    int n_tasks;
    ucc_context_t *ctx;
    ucc_coll_task_t *tasks[UCC_SCHEDULE_MAX_TASKS];
} ucc_schedule_t;

ucc_status_t ucc_event_manager_notify(ucc_coll_task_t *parent_task,
                                      ucc_event_t event);

void ucc_schedule_add_task(ucc_schedule_t *schedule, ucc_coll_task_t *task);

ucc_status_t ucc_schedule_start(ucc_schedule_t *schedule) {
    schedule->n_completed_tasks  = 0;
    schedule->super.super.status = UCC_INPROGRESS;
    return ucc_event_manager_notify(&schedule->super,
                                    UCC_EVENT_SCHEDULE_STARTED);
}

void ucc_event_manager_subscribe(ucc_event_manager_t *em,
                                 ucc_event_t event,
                                 ucc_coll_task_t *task,
                                 ucc_task_event_handler_p handler);
```

CL/HIER, rab allreduce

```
ucc_status_t ucc_cl_hier_allreduce_rab_init(ucc_base_coll_args_t *coll_args,
                                            ucc_base_team_t *team,
                                            ucc_coll_task_t **task)
{
    ucc_schedule_t *schedule;
    ucc_coll_task_t tasks[3];
 ...
    schedule = &ucc_cl_hier_get_schedule(cl_team)->super.super;
 ...
    task[0] = initialize_task_from_node_sbgp_tl(coll_args, TASK_REDUCE);
    task[1] = initialize_task_from_node_leaders_sbgp_tl(coll_args, TASK_ALLREDUCE);
    task[2] = initialize_task_from_node_sbgp_tl(coll_args, TASK_BCAST);

    ucc_event_manager_subscribe(&schedule->super.em, UCC_EVENT_SCHEDULE_STARTED,
                                tasks[0], ucc_task_start_handler);

    for (i = 1; i < n_tasks; i++) {
        ucc_event_manager_subscribe(&tasks[i - 1]->em, UCC_EVENT_COMPLETED,
                                    &tasks[i], ucc_task_start_handler);
        ucc_schedule_add_task(schedule, tasks[i]);
    }
}

static ucc_status_t ucc_cl_hier_allreduce_rab_start(ucc_coll_task_t *task)
{
    ucc_schedule_t *schedule = ucc_derived_of(task, ucc_schedule_t);
    return ucc_schedule_start(schedule);
}
```

Runtime example

- Enable 2 CLs basic and hier (selection depends on the coll args and uses score_map)
- User can specify which TLs can be used at different subgrouping levels in CL/HIER
  - When several TLs are allowed per sbgp then the selection of task for a schedule is done using score_map again

- nnodes=32; ppn=40;
- mpirun
  - -x UCC_CL_HIER_NODE_LEADERS_SBGP_TLS=ucp,sharp
  - -x UCC_CL_HIER_TLS=ucp,sharp
  - -x UCC_CL_BASIC_TLS=ucp
  - -x UCC_CLS=basic,hier
  - -np $((nnodes*ppn)) --map-by ppr:$ppn:node --bind-to core ./install/bin/ucc_perftest -c allreduce -b 1 -e 512 -w 5000 -n 10000

msgsize, B	CL/BASIC	HIER	HIER+SHARP
4	15.38	13.48	11.03
8	15.26	14.9	12.03
16	15.24	14.72	11.72
32	16.47	17.65	13.21
64	18.63	16.34	14.32
128	23.75	21.55	18.19
256	25.77	27.92	22.88
512	32.07	34.42	23.81
1024	60.35	33.06	28.49
2048	82.8	37.01	31.88

UCC GPU Collectives

GPU Collectives Concepts

- Support for GPU memory
  - Source or Destination buffer resides on GPU
  - Operations with GPU datatypes (float16, bfloat16)
  - Reductions using GPU kernel

- Support for GPU programming model
  - Streams and ordered execution
  - Events and synchronization

GPU Memory

- User sets memory type for input and output buffers or specify UCC_MEMORY_TYPE_UNKNOWN to do memory type detection
- Collective is passed to one of CL/TL that supports GPU memory through score function
- TLs might use UCC Memory Component to do some local operations with memory (memory type detection, memory allocation, reduction)
- All TLs in UCC 1.0 supports CUDA memory

```
typedef struct ucc_coll_buffer_info {
    void              *buffer;   /*!< Starting address of the send/recv buffer */
    ucc_count_t        count;    /*!< Total number of elements in the buffer */
    ucc_datatype_t     datatype; /*!< Datatype of each buffer element */
    ucc_memory_type_t mem_type;  /*!< Memory type of buffer as defined by @ref
                                      ucc_memory_type */
} ucc_coll_buffer_info_t;
```

```
typedef struct ucc_coll_args {
    uint64_t                        mask;
    ucc_coll_type_t                 coll_type; /*!< Type of collective operation */
    union {
        ucc_coll_buffer_info_t      info;   /*!< Buffer info for the collective */
        ucc_coll_buffer_info_v_t    info_v; /*!< Buffer info for the collective */
    } src;
    union {
        ucc_coll_buffer_info_t      info;   /*!< Buffer info for the collective */
        ucc_coll_buffer_info_v_t    info_v; /*!< Buffer info for the collective */
    } dst;
    struct {
        ucc_reduction_op_t          predefined_op; /*!< Reduction operation, if
                                                        reduce or all-reduce
                                                        operation selected */
        void                       *custom_op; /*!< User defined
                                                    reduction operation */
        void                       *custom_dtype;
    } reduce;
    uint64_t                        flags;
    uint64_t                        root; /*!< Root endpoint for rooted
                                               collectives */
    ucc_error_type_t                error_type; /*!< Error type */
    ucc_coll_id_t                   tag; /*!< Used for ordering collectives */
    ucc_coll_callback_t             cb;
    double                          timeout; /*!< Timeout in seconds */
} ucc_coll_args_t;
```

UCC Memory Component

- For each memory type UCC provides component that implements base set of operations
- Components are loaded and initialized at runtime

```
typedef struct ucc_mc_ops {
    ucc_status_t (*mem_query)(const void *ptr, ucc_mem_attr_t *mem_attr);
    ucc_status_t (*mem_alloc)(ucc_mc_buffer_header_t **h_ptr, size_t size);
    ucc_status_t (*mem_free)(ucc_mc_buffer_header_t *h_ptr);
    ucc_status_t (*reduce)(const void *src1, const void *src2, void *dst,
                           size_t count, ucc_datatype_t dt,
                           ucc_reduction_op_t op);
    ucc_status_t (*reduce_multi)(const void *src1, const void *src2, void *dst,
                                 size_t n_vectors, size_t count, size_t stride,
                                 ucc_datatype_t dt, ucc_reduction_op_t op);
    ucc_status_t (*memcpy)(void *dst, const void *src, size_t len,
                           ucc_memory_type_t dst_mem,
                           ucc_memory_type_t src_mem);
    ucc_status_t (*flush)();
 } ucc_mc_ops_t;
```

CUDA Programming Model with UCC

- One of the key concept in CUDA programming is CUDA stream
  - Operations issued to the same stream will execute in order
  - Operations issued to separate stream have no dependency and might execute in any order

- Collective might be considered as CUDA operation, and it should follow stream semantics

```
cudaStream_t strm;
ucc_coll_args_t args;
...
cuda_compute_kernel1<<<32, 32, 0, strm>>>(compute_input_buf,
                                          compute_output_buf);

args.coll_type       = UCC_COLL_TYPE_ALLREDUCE;
args.src.info.buffer = compute_output_buf;
args.dst.info.buffer = coll_output_buf;
...
ucc_collective_init(&args, &req, team);
ucc_collective_post(req);

cuda_compute_kernel2<<<32, 32, 0, strm>>>(coll_output_buf);
```

CUDA Programming Model with UCC

- One of the key concept in CUDA programming is CUDA stream
  - Operations issued to the same stream will execute in order
  - Operations issued to separate stream have no dependency and might execute in any order

- Collective might be considered as CUDA operation, and it should follow stream semantics

```
cudaStream_t strm;
ucc_coll_args_t args;
...
cuda_compute_kernel1<<<32, 32, 0, strm>>>(compute_input_buf,
                                          compute_output_buf);

args.coll_type       = UCC_COLL_TYPE_ALLREDUCE;
args.src.info.buffer = compute_output_buf;
args.dst.info.buffer = coll_output_buf;
...
ucc_collective_init(&args, &req, team);
ucc_collective_post(req);

cuda_compute_kernel2<<<32, 32, 0, strm>>>(coll_output_buf);
```

compute_output_buf is not ready yet

allreduce still in progress

CUDA Programming Model with UCC

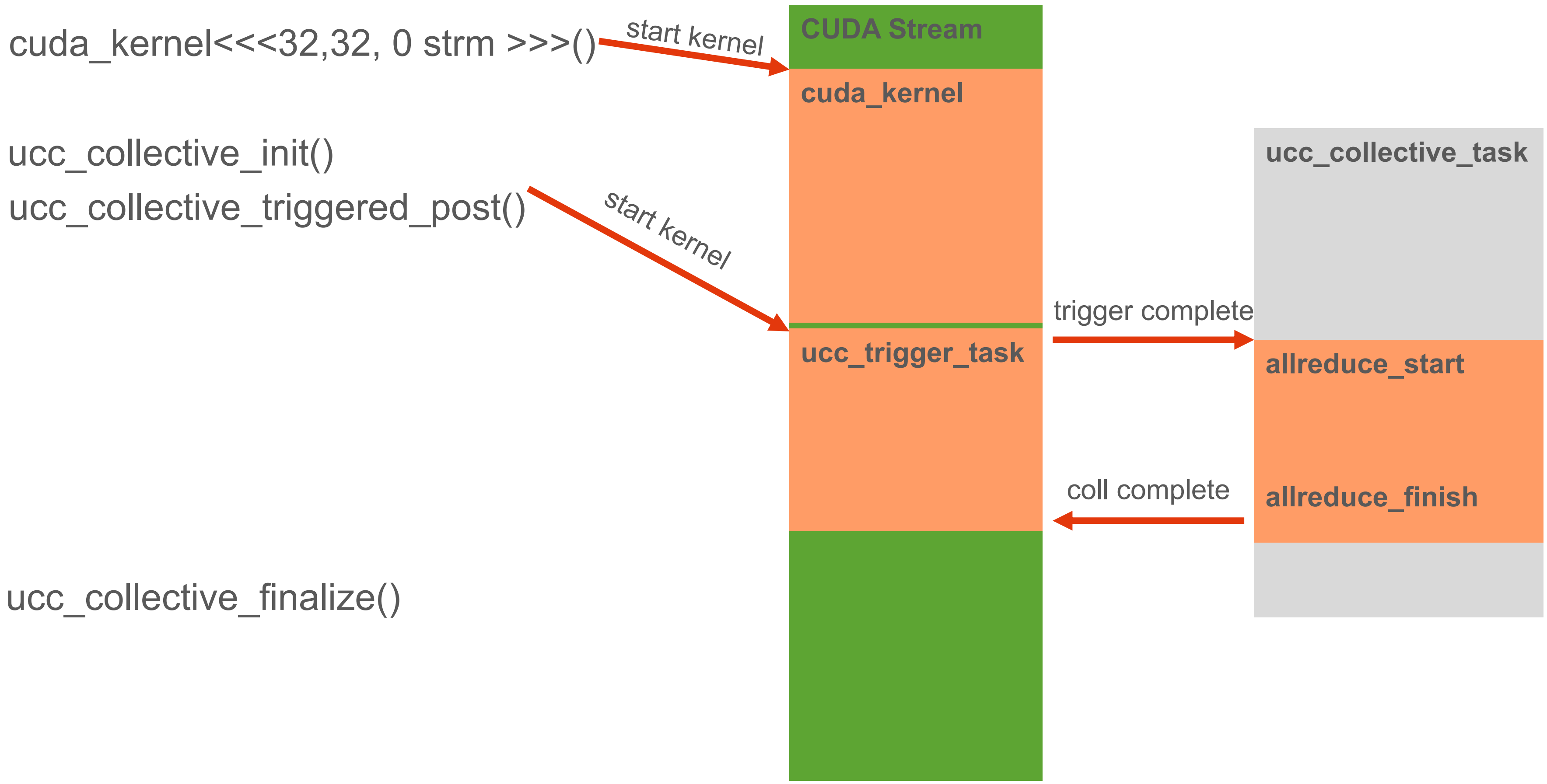
- UCC API supports CUDA streams as part of triggered collectives
  - CUDA stream is represented by UCC Execution Engine
  - Collective posted to UCC CUDA Execution Engine effectively issued into stream

```
cudaStream_t strm;
ucc_coll_args_t args;
ucc_ee_params_t ee_params;
ucc_ee_h cuda_ee;
ucc_ev_t comp_ev;

ee_params.ee_type         = UCC_EE_CUDA_STREAM;
ee_params.ee_context      = strm;
ee_params.ee_context_size = sizeof(cudaStream_t);
ucc_ee_create(team, &ee_params, &cuda_ee),
...
cuda_compute_kernel1<<<32, 32, 0, strm>>>(compute_input_buf,
                                          compute_output_buf);
args.coll_type       = UCC_COLL_TYPE_ALLREDUCE;
args.src.info.buffer = compute_output_buf;
args.dst.info.buffer = coll_output_buf;
...
ucc_collective_init(&args, &req, team);
comp_ev.ev_type = UCC_EVENT_COMPUTE_COMPLETE;
comp_ev.req     = req;
ucc_collective_triggered_post(ee, comp_ev);

cuda_compute_kernel2<<<32, 32, 0, strm>>>(coll_output_buf);
```

UCC Triggered Post in TLs

- TL NCCL supports triggered operations with CUDA stream natively
- TL UCP and TL SHARP requires additional logic to guarantee correct execution order
  - Input dependency – start collective only after all previously submitted stream work is done
  - Output dependency – mark stream busy while collective is in progress
  - Issue all CUDA work to stream when return from non-blocking UCC function

UCC Triggered Task



Future work

- UCC TL CUDA
  - GPU topology aware
  - Use of CUDA primitives for communication between peers on local node
  - Support for allreduce and reduce_scatter for ucc_triggered_post
  - TL/CUDA: implement tl cuda by Sergei-Lebedev · Pull Request #336 · openucx/ucc (github.com)

UCC One-sided/RMA Collectives

One-sided Collectives

- One-sided collectives leverage one-sided RMA operations to perform collectives over a UCC team
  - Allows for loose synchronization on collective start and completion
  - Directly maps operations to hardware primitives

- Relationship to PGAS programming models
  - Focus on OpenSHMEM

Mapping from the Model to UCC: Memory Segments

- Directly map pre-allocated memory from user to UCC Context
  - Additional UCC Context parameter: ucc_mem_map_params_t
  - E.g., OpenSHMEM's Symmetric heap
- ucc_mem_map_t
  - Binds user allocated memory to a UCC Context
  - Memory usable by multiple UCC Teams with shared context
- One-sided collectives operate on user provided buffers

```
/**
 *
 *  @ingroup UCC_CONTEXT_DT
 */
typedef struct ucc_mem_map {
    void *   address; /*!< the address of a buffer to be attached to a UCC context */
    size_t   len;     /*!< the length of the buffer */
} ucc_mem_map_t;

/**
 *
 * @ingroup UCC_CONTEXT_DT
 */
typedef struct ucc_mem_map_params {
    ucc_mem_map_t *segments;   /*!< array of ucc_mem_map elements */
    uint64_t       n_segments; /*!< the number of ucc_mem_map elements */
} ucc_mem_map_params_t;
```

Example: UCC Context Creation with Memory Parameters

```
int                    num_maps = 0;
ucc_context_h          ucc_context;
ucc_context_config_h ctx_config;
ucc_context_params_t ctx_params;
ucc_mem_map_t          maps[3];

...

ctx_params.mask =
    UCC_CONTEXT_PARAMS_FIELD_OOB | UCC_CONTEXT_PARAM_FIELD_MEM_PARAMS;
ctx_params.oob.allgather = oob_allgather;
ctx_params.oob.req_test  = oob_allgather_test;
ctx_params.oob.req_free  = oob_allgather_free;
ctx_params.oob.coll_info = (void *)oshmem_comm_world;
ctx_params.oob.n_oob_eps = ompi_comm_size(oshmem_comm_world);
ctx_params.oob.oob_ep    = ompi_comm_rank(oshmem_comm_world);

for (num_maps = 0; num_maps < 3; num_maps++) {
        maps[num_maps].address = segment[num_maps].base_address;
        maps[num_maps].len = segment[num_maps].length;
}
ctx_params.mem_params.segments = maps;
ctx_params.mem_params.n_segments = 3;

...

if (UCC_OK !=
    ucc_context_create(ucc_lib, &ctx_params, ctx_config, &ucc_context)) {
        fprintf(stderr, "ucc context creation failed\n");
        goto cleanup;
}
...
```

- Example assumes 3 memory segments
  - Following OpenSHMEM memory model with an additional heap for atomics

Additional Parameters

Calling One-sided Algorithm

- One-sided collectives require scratch synchronization buffer for completion semantics
  - Two methods of allocating:
    1. User allocated buffer passed as argument to collective
    2. Team-based buffer allocated by UCC
- User allocated buffers:
  - On UCC Team creation, pass UCC_TEAM_FLAG_COLL_WORK_BUFFER flag
  - Additional UCC context query option: UCC_CONTEXT_ATTR_FIELD_WORK_BUFFER_SIZE
  - Returns size required for UCC one-sided collectives

```
long                        *work_buffer;
ucc_context_attr_t           attr;

attr.mask = UCC_CONTEXT_ATTR_FIELD_WORK_BUFFER_SIZE;
ucc_context_get_attr(ucc_context, &attr);
size = attr.global_work_buffer_size;

work_buffer = (long *)malloc(size);
```

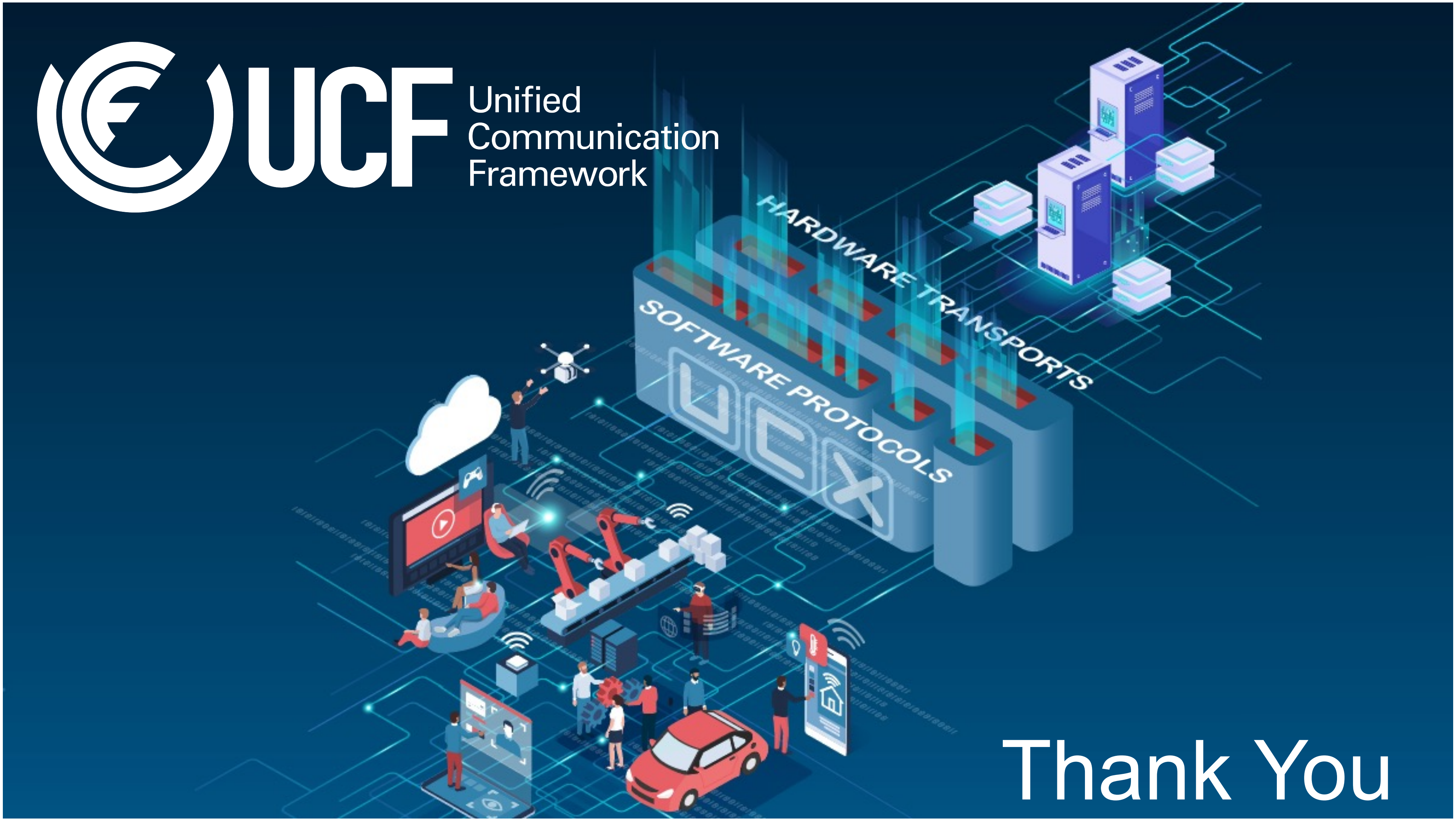
- Example Algorithm: Preliminary A2A algorithm (PR #323)
  - Basic algorithm (see right)

```
for each rank in team:
      put(src, dest[myrank * size], rank)
      atomic_inc(work_buffer, rank)

wait until completion
```

Example Call of One-sided A2A

```
ucc_coll_args_t coll = {
    .mask = UCC_COLL_ARGS_FIELD_GLOBAL_WORK_BUFFER | UCC_COLL_ARGS_FIELD_FLAGS,
    .coll_type          = UCC_COLL_TYPE_ALLTOALL,
    .src.info           = {
                 .buffer   = (void *)sbuf,
                 .count    = count * proc_count,
                 .datatype = UCC_DT_INT64,
                 .mem_type = UCC_MEMORY_TYPE_UNKNOWN},
    .dst.info           = {
                 .buffer   = rbuf,
                 .count    = count * proc_count,
                 .datatype = UCC_DT_INT64,
                 .mem_type = UCC_MEMORY_TYPE_UNKNOWN},
    .global_work_buffer = work_buffer,
    .flags              = UCC_COLL_ARGS_FLAG_MEM_MAPPED_BUFFERS,
};

ucc_collective_init(&coll, &req, my_ucc_team);
...
```

- Call for invoking one-sided A2A
  - Env. Variable: UCC_TL_UCP_TUNE=alltoall:0-inf:@1


UCF
Unified
Communication
Framework
HARDWARE TRANSPORTS
SOFTWARE PROTOCOLS
UCX
Thank You