Unified Collective Communications (UCC) Specification

Version 1.4

Unified Collective
Communication

Contents

Chapter 1

Main Page

UCC is a collective communication operations API and library that is flexible, complete, and feature-rich for current and emerging programming models and runtimes.

Chapter 2

Design

- Highly scalable and performant collectives for HPC, AI/ML and I/O workloads
- Nonblocking collective operations that cover a variety of programming models
- Flexible resource allocation model
- Support for relaxed ordering model
- Flexible synchronous model
- Repetitive collective operations (init once and invoke multiple times)
- Hardware collectives are a first-class citizen

Component Diagram

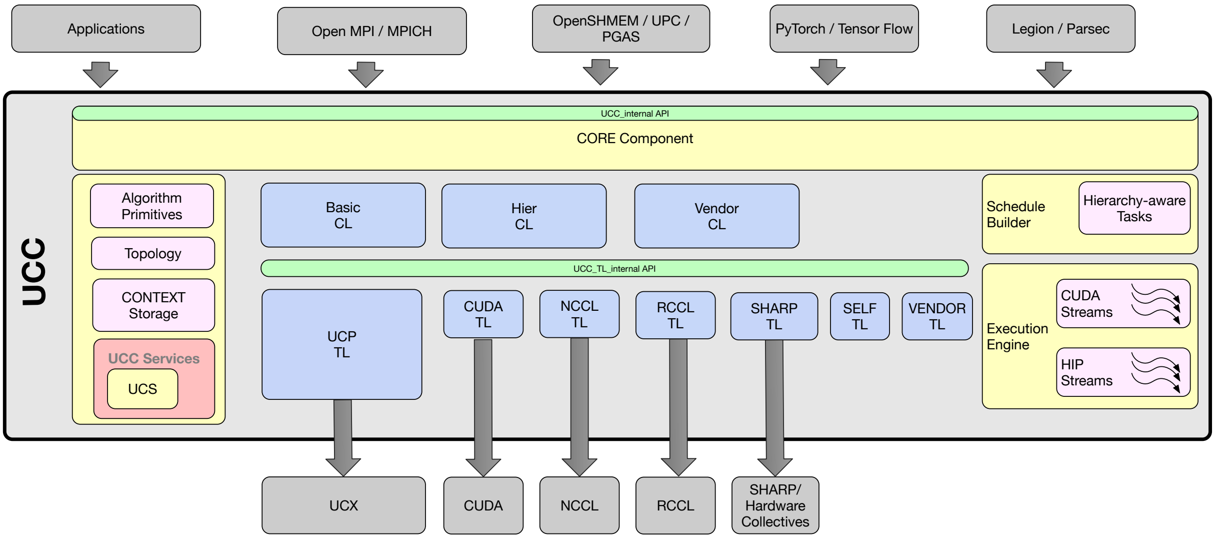


Figure 2.1: UCC Components and Usage

Chapter 3

Library Initialization and Finalization

These routines are responsible for allocating, initializing, and finalizing the resources for the library.

The UCC can be configured in three thread modes UCC_THREAD_SINGLE, UCC_THREAD_FUNNEL↩ ED, and UCC_LIB_THREAD_MULTIPLE. In the UCC_THREAD_SINGLE mode, the user program must not be multithreaded. In the UCC_THREAD_FUNNELED mode, the user program may be multithreaded. However, all UCC interfaces should be invoked from the same thread. In the UCC_THREAD_MULTIPLE mode, the user program can be multithreaded and any thread may invoke the UCC operations.

The user can request different types of collective operations that vary in their synchronization models. The valid synchronization models are UCC_NO_SYNC_COLLECTIVES and UCC_SYNC_COLLECTIVES. The details of these synchronization models are described in the collective operation section.

The user can request the different collective operations and reduction operations required. The complete set of valid collective operations and reduction types are defined with the structures ucc_coll_type_t and ucc_reduction_op_t.

Chapter 4

Communication Context

The ucc_context_h is a communication context handle. It can encapsulate resources required for collective operations on team handles. The contexts are created by the ucc_context_create operation and destroyed by the ucc_context_destroy operation. The create operation takes in user-configured ucc_context_params_t structure to customize the context handle. The attributes of the context created can be queried using the ucc_context_get_attribs operation.

When no out-of-band operation (OOB) is provided, the ucc_context_create operation is local requiring no communication with other participants. When OOB operation is provided, all participants of the OOB operation should participate in the create operation. If the context operation is a collective operation, the ucc_context_destroy operation is also a collective operation .i.e., all participants should call the destroy operation.

The context can be created as an exclusive type or shared type by passing constants UCC_CONTEXT_EXCLUSIVE and UCC_CONTEXT_SHARED respectively to the ucc_context_params_t structure. When context is created as a shared type, the same context handle can be used to create multiple teams. When context is created as an exclusive type, the context can be used to create multiple teams but the team handles cannot be valid at the same time; a valid team is defined as a team object where the user can post collective operations.

Notes : From the user perspective, the context handle represents a communication resource. The user can create one context and use it for multiple teams or use with a single team. This provides a finer control of resources for the user. From the library implementation perspective, the context could represent the network parallelism. The UCC library implementation can choose to abstract injection queues, network endpoints, GPU device context, UCP worker, or UCP endpoints using the communication context handles.

Chapter 5

Teams

The ucc_team_h is a team handle, which encapsulates the resources required for group operations such as collective communication operations. The participants of the group operations can either be an OS process, a control thread or a task.

Create and destroy routines: ucc_team_create_post routine is used to create the team handle and ucc_team_create_test routine for learning the status of the create operation. The team handle is destroyed by the ucc_team_destroy operation. A team handle is customized using the user configured ucc_team_params_t structure.

Invocation semantics: The ucc_team_create_post is a nonblocking collective operation, in which the participants are determined by the user-provided OOB collective operation. Overlapping of multiple ucc_team_create_post operations are invalid. Posting a collective operation before the team handle is created is invalid. The team handle is destroyed by a blocking collective operation; the participants of this collective operation are the same as the create operation. When the user does not provide an OOB collective operation, all participants calling the ucc_create_post operation will be part of a new team created.

Communication Contexts: Each process or a thread participating in the team creation operation contributes one or more communication contexts to the operation. The number of contexts provided by all participants should be the same and each participant should provide the same type of context. The newly created team uses the context for collective operations. If the communication context abstracts the resources for the library, the collective operations on this team uses the resources provided by the context.

Endpoints: That participants to the ucc_team_create_post operation can provide an endpoint, a 64-bit unsigned integer. The endpoint is an address for communication. Each participant of the team has a unique integer as endpoint .i.e., the participants of the team do not share the same endpoint. For example, the user can bind the endpoint to the parallel programming model's index such as OpenSHMEM PE, an OS process ID, or a thread ID. The UCC implementation can use the endpoint as an index to identify the resources required for communication such as communication contexts. When the user does not provide the endpoint, the library generates the endpoint, which can be queried by the user. In addition to the endpoint, the user can provide information about the endpoints such as whether the endpoint is a continuous range or not.

Ordering: The collective operations on the team can either be ordered or unordered. In the ordered model, the UCC collectives are invoked in order .i.e., on a given team, each of the participants of the collective operation invokes the operation in the same order. In the unordered model, the collective operations are not necessarily invoked in the same order.

Interaction with Threads: The team can be created in either mode .i.e., the library initialized by UCC_LIB_THREAD_MULTIPLE, UCC_LIB_THREAD_SINGLE, or UCC_LIB_THREAD_FUNNEDLED. In the UCC_LIB_THREAD_MULTIPLE mode, each of the user threads can post a collective operation. However, it is not valid to post concurrent collectives operations from multiple threads to the same team.

Memory per Team: A team can be configured by a memory descriptor described by ucc_mem_map_params_t structure. The memory can be used as an input and output buffers for the collective operation. This is particularly useful for PGAS programming models, where the input and output buffers are defined before the invocation operation. For example, the input and output buffers in the OpenSHMEM programming model are defined during the programming model initialization.

Synchronization Model: The team can be configured to support either synchronized collectives or non-synchronized collectives. If the UCC library is configured with synchronized collective operations and the team is configured with non-synchronized collective operations, the library might not be able to provide any optimizations and might support only synchronized collective operations.

Outstanding Calls: The user can configure maximum number of outstanding collective operations of any type for a given team. This is represented by an unsigned integer. This is provided as a hint to the library for resource management.

Team ID: The team identifier is a unique 64-bit unsigned integer for the given process .i.e, the team identifier should be unique for all teams it creates or participates. If the team identifier is provided by the user, it should be passed as a configuration parameter to the team create operation.

Split Team Operations

The team split routines provide an alternate way to create teams. All split routines require a parent team and all participants of the parent team call the split operation. The participants of the new team may include some or all participants of the parent team.

The newly created team shares the communication contexts with the parent team. The endpoint of the new team is contiguous and is not related to the parent team. It inherits the thread model, synchronization model, collective ordering model, outstanding collectives configuration, and memory descriptor from the parent team.

The split operation can be called by multiple threads, if the parent team to the split operations are different and if it agrees with the thread model of the UCC library.

Notes: The rationale behind requiring all participants of the parent team to participate in the split operation is to avoid overlapping participants between multiple split operations, which is known to increase the implementation complexity. Also, currently, higher-level programming models do not require these semantics.

Chapter 6

Types of Collective Operations

A UCC collective operation is a group communication operation among the participants of the team. All participants of the team are required to call the collective operation. Each participant is represented by the endpoint that is unique to the team used for the collective operation. This section provides a set of routines for launching, progressing, and completing the collective operations.

Invocation semantics: The ucc_collective_init routine is a non-blocking collective operation to initialize the buffers, operation type, reduction type, and other information required for the collective operation. All participants of the team should call the initialize operation. The collective operation is invoked using a ucc_collective_post operation. ucc_collective_init_and_post operation initializes as well as post the collective operation.

Collective Type: The collective operation supported by UCC is defined by the enumeration ucc_coll_type_t. The semantics are briefly described here, however in most cases it agrees with the semantics of collective operations in the popular programming models such as MPI and OpenSHMEM. When they differ, the semantics changes are documented. All collective operations execute on the team. For the collective operations defined by ucc_coll_type_t, all participants of the team are required to participate in the collective operations. Further the team should be created with endpoints, where the "eps" should be ordered and contiguous.

UCC supports three types of collective operations: (a) UCC_{ALLTOALL, ALLTOALLV, ALLGATHER, ALLGATHERV, ALLREDUCE, REDUCE_SCATTER, REDUCE_SCATTERV, BARRIER} operations where all participants contribute to the results and receive the results (b) UCC_{REDUCE, GATHER, GATHERV, FANIN} where all participants contribute to the result and one participant receives the result. The participant receiving the result is designated as root. (c) UCC_{BROADCAST, SCATTER, SCATTERV, FANOUT} where one participant contributes to the result, and all participants receive the result. The participant contributing to the result is designated as root.

- The UCC_COLL_TYPE_BCAST operation moves the data from the root participant to all participants in the team.
- The UCC_COLL_TYPE_BARRIER synchronizes all participants of the collective operation. In this routine, first, each participant waits for all other participants to enter the operation. Then, once it learns the entry of all other participants into the operation, it exits the operation completing it locally.
- In the UCC_COLL_TYPE_FAN_IN operation, the root participant synchronizes with all participants of the team. The non-root completes when it sends synchronizing message to the root. Unlike UCC_COLL_TYPE_BARRIER, it doesn't have to synchronize with the rest of the non-root participants. The root participant completes the operation when it receives synchronizing messages from all non-root participants of the team.
- The UCC_COLL_TYPE_FAN_OUT operation is a synchronizing operation like UCC_COLL_TYPE_FAN_OUT. In this operation, the root participant sends a synchronizing message to all non-root participants and completes. The non-root participant completes once it receives a message from the root participant.
- In the UCC_COLL_TYPE_GATHER operation, each participant of the collective operation sends data to the root participant. All participants send the same amount of data (block_size) to the root. The

size of the block is “dt_elem_size ∗ count”. The total amount of data received by the root is equal to block_size ∗ num_participants. Here, the “count” represents the number of data elements. The "dt_elem_size" represents the size of the data element in bytes. The "num_participants" represents the number of participants in the team. The data on the root is placed in the receive buffer ordered by the “ep” ordering. For example, if the participants’ endpoints are ordered as “ep_a” to “ep_n”, the data from the participant with ep_i is placed as an “ith” block on the receive buffer.

- The UCC_COLL_TYPE_ALLGATHER operation is similar to UCC_COLL_TYPE_GATHER with one exception. Unlike in GATHER operation, the result is available at all participants’ receive buffer instead of only at the root participant.

  ```
  Each participant sends the data of size "block_size" to all other participants
  ```

  in the collective operation. The size of the block is “dt_elem_size ∗ count”. Here, the “count” represents the number of data elements. The "dt_elem_size" represents the size of the data element in bytes. The data on each participant is placed in the receive buffer ordered by the “ep” ordering. For example, if the participants’ endpoints are ordered as “ep_a” to “ep_n”, the data from the participant with ep_i is placed as an “ith” block on the receive buffer.

- In the UCC_COLL_TYPE_SCATTER operation, the root participant of the collective operation sends data to all other participants. It sends the same amount of data (block_size) to all participants. The size of the block (block_size) is “dt_elem_size ∗ count”. The total amount of data sent by the root is equal to block_size ∗ num_participants. Here, the “count” represents the number of data elements. The "dt_elem_size" represents the size of the data element in bytes. The "num_participants" represents the number of participants in the team.

- In the UCC_COLL_TYPE_ALLTOALL collective operation, all participants exchange a fixed amount of the data. For a given participant, the size of data in src buffer is “size”, where size is dt_elem_size ∗ count ∗ num_participants. Here, the “count” represents the number of data elements per destination. The "dt_elem_size" represents the size of the data element in bytes. The "num_participants" represents the number of participants in the team. The size of src buffer is the same as the dest buffer, and it is the same across all participants. Each participant exchanges “dt_elem_size ∗ count “ data with every participant of the collective.

- In UCC_COLL_TYPE_REDUCE collective the element-wise reduction operation is performed on the src buffer of all participants in the collective operation. The result is stored on the dst buffer of the root. The size of src buffer and dst buffer is the same, which is equal to “dt_elem_size ∗ count”. Here, the “count” represents the number of data elements. The "dt_elem_size" represents the size of the data element in bytes.

- The UCC_COLL_TYPE_ALLREDUCE first performs an element-wise reduction on the src buffers of all participants. Then the result is distributed to all participants. After the operation, the results are available on the dst buffer of all participants. The size of src buffer and dst buffer is the same for all participants. The size of src buffer and dst buffer is the same, which is equal to “dt_elem_size ∗ count”. Here, the “count” represents the number of data elements. The "dt_elem_size" represents the size of the data element in bytes.

- The UCC_COLL_TYPE_REDUCE_SCATTER first performs an element-wise reduction on the src buffer and then scatters the result to the dst buffer. The "size" of src buffer is “count ∗ dt_elem_size”, where dt_elem_size is the number of bytes for the data type element and count is the number of elements of that datatype. It is the user’s responsibility to ensure that data and the result are equally divisible among the participants. Assuming that the result is divided into “n” blocks, the ith block is placed in the receive buffer of endpoint “i”. Like other collectives, for this collective, the “ep” should be ordered and contiguous.

INPLACE: When INPLACE is set for UCC_COLL_TYPE_REDUCE_SCATTER, UCC_COLL_TYPE_REDUCE, UCC_COLL_TYPE_ALLREDUCE, UCC_COLL_TYPE_SCATTER, and UCC_COLL_TYPE_ALLTOALL the receive buffers act as both send and receive buffer.

For UCC_COLL_TYPE_BCAST operation, setting INPLACE flag has no impact.

The "v" Variant Collective Types: The UCC_COLL_TYPE_{ALLTOALLV, SCATTERV, GATHERV, and REDUCE_SCATTERV} operations add flexibility to their counter parts (.i.e., ALLTOALL, SCATTER, GATHER, and REDUCE_SCATTER) in that the location of data for the send and receive are specified by displacement arrays.

Reduction Types: The reduction operation supported by UCC_{ALLREDUCE, REDUCE, REDUCE_SC↩ATTER, REDUCE_SCATTERV} operation is defined by the enumeration ucc_reduction_op_t. The valid datatypes for the reduction is defined by the enumeration ucc_datatype_t.

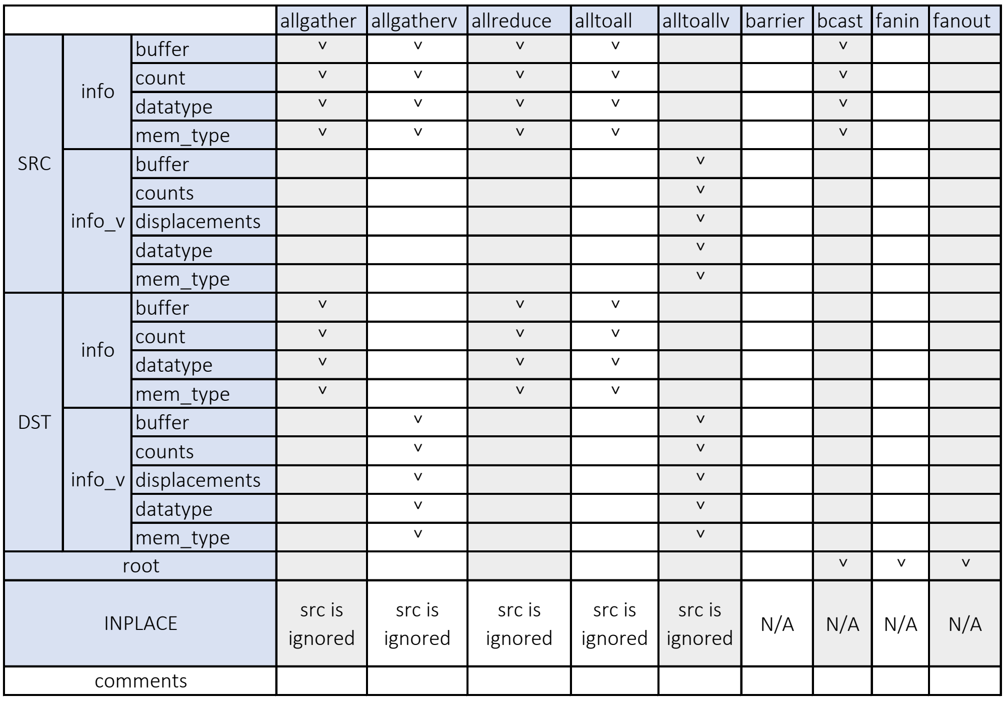
Ordering: The team can be configured for ordered collective operations or unordered collective operations. For unordered collectives, the user is required to provide the "tag", which is an unsigned 64-bit integer.

Synchronized and Non-Synchronized Collectives: In the synchronized collective model, on entry, the participants cannot read or write to other participants without ensuring all participants have entered the collective operation. On the exit of the collective operation, the participants may exit after all participants have completed the reading or writing to the buffers.

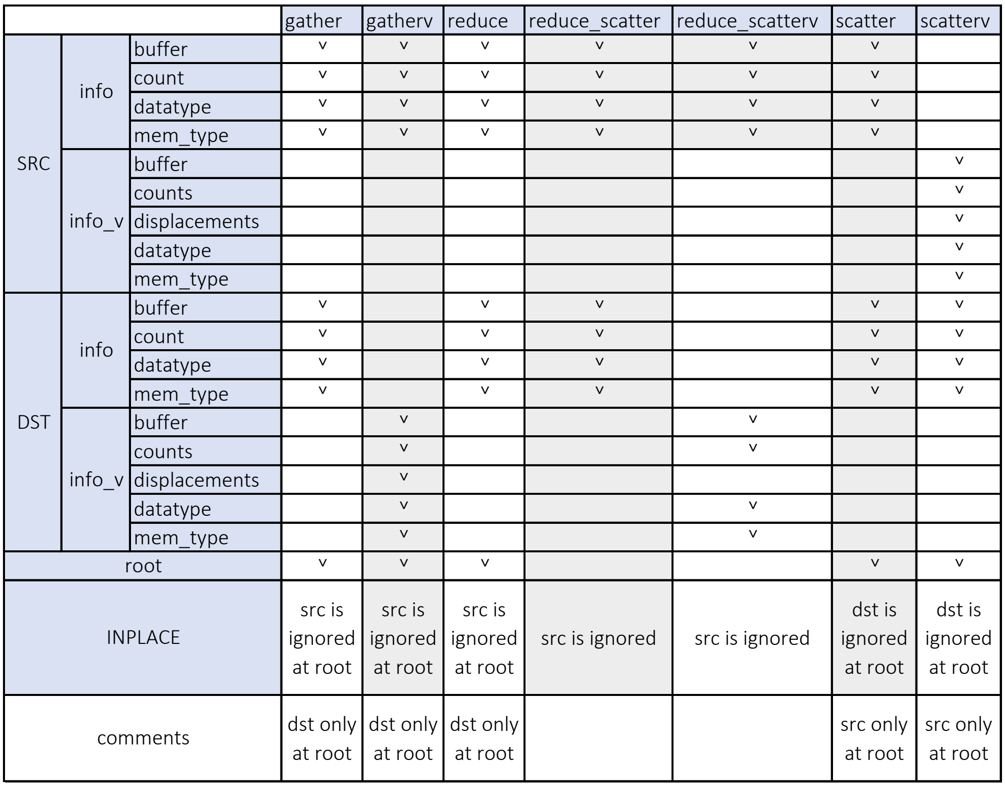
In the non-synchronized collective model, on entry, the participants can read or write to other participants. If the input and output buffers are defined on the team and RMA operations are used for data transfer, it is the responsibility of the user to ensure the readiness of the buffer. On exit, the participants may exit once the read and write to the local buffers are completed.

Buffer Ownership: The ownership of input and output buffers are transferred from the user to the library after invoking the ucc_collective_init routine. On return from the routine, the ownership is transferred back to the user on ucc_collective_finalize. However, after invoking and returning from ucc_collective_post or ucc_collective_init_and_post routines, the ownership stays with the library and it is returned to the user, when the collective is completed.

The table below lists the necessary fields that user must initialize depending on the collective operation type.



			allgather	allgatherv	allreduce	alltoall	alltoallv	barrier	bcast	fanin	fanout
SRC	info	buffer	v	v	v	v			v		
		count	v	v	v	v			v		
		datatype	v	v	v	v			v		
		mem_type	v	v	v	v			v		
	info_v	buffer					v				
		counts					v				
		displacements					v				
		datatype					v				
		mem_type					v				
DST	info	buffer	v		v	v					
		count	v		v	v					
		datatype	v		v	v					
		mem_type	v		v	v					
	info_v	buffer		v			v				
		counts		v			v				
		displacements		v			v				
		datatype		v			v				
		mem_type		v			v				
root									v	v	v
INPLACE			src is ignored	src is ignored	src is ignored	src is ignored	src is ignored	N/A	N/A	N/A	N/A
comments											



			gather	gatherv	reduce	reduce_scatter	reduce_scatterv	scatter	scatterv
SRC	info	buffer	v	v	v	v	v	v	
		count	v	v	v	v	v	v	
		datatype	v	v	v	v	v	v	
		mem_type	v	v	v	v	v	v	
	info_v	buffer							v
		counts							v
		displacements							v
		datatype							v
		mem_type							v
DST	info	buffer	v		v	v		v	v
		count	v		v	v		v	v
		datatype	v		v	v		v	v
		mem_type	v		v	v		v	v
	info_v	buffer		v			v		
		counts		v			v		
		displacements		v					
		datatype		v			v		
		mem_type		v			v		
root			v	v	v			v	v
INPLACE			src is ignored at root	src is ignored at root	src is ignored at root	src is ignored	src is ignored	dst is ignored at root	dst is ignored at root
comments			dst only at root	dst only at root	dst only at root			src only at root	src only at root

Chapter 7

Execution Engine and Events

The execution engine is an execution context that supports event-driven network execution on the CU↩ DA streams, CPU threads, and DPU threads. It is intended to interact with execution threads that are asynchronous (offloaded collective execution) which can be implemented on GPUs, DPUs, or remote CPUs.

UCC supports triggering collective operations by library-generated and user-generated events. The library events are generated on posting or completion of operations. The user-generated events include the completion of compute or communication operations. With a combination of library-generated and user-generated events, one can build dependencies between compute and collective operations, or between the collective operations.

Besides the execution engine, events are key for event-driven execution. The operations on the execution engines generate events that are stored internally on the execution engines. The valid events are defined by ucc_event_type_t. If the underlying hardware doesn't support event-driven execution, the implementations can implement this with the event queues or lists.

The interaction between the user and library is through the UCC interfaces. ucc_ee_create creates execution engines. The user or library can generate an event and post it to the execution engines using ucc_ee_set↩ _event interface. The user can wait on the events with the ucc_ee_wait interface. The user can get the event from the ee using ucc_ee_get_event interface and acknowledge the event with ucc_ee_ack_event interface. Once acknowledged, the library destroys the event.

Thread Mode: While in the UCC_THREAD_MULTIPLE mode, the execution engine and operations can be invoked from multiple threads.

Order: All non-triggered operations posted to the execution engine are executed in-order. However, there are no ordering guarantees between the execution engines.

Triggered Operations

Triggered operations enable the posting of operations on an event. For triggered operations, the team should be configured with event-driven execution. The collection operations is defined by the interface ucc_collective↩ _triggered_post.

The operations are launched on the event. So, there is no order established by the library. If user desires an order for the triggered operations, the user should provide the tag for matching the collective operations.

Interaction between an User Thread and Event-driven UCC

The figure shows the interaction between application threads and the UCC library configured with event-driven teams. In this example scenario, we assume that the UCC team are configured with two events queues - one for post operations and one for completions.

(1) The application initializes the collective operation when it knows the control parameters of the collective such as buffer addresses, lengths, and participants of the collective. The data need not be ready as it posts the collective operation which will be triggered on an event. For example, the event here is the completion of

compute by the application.

(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.

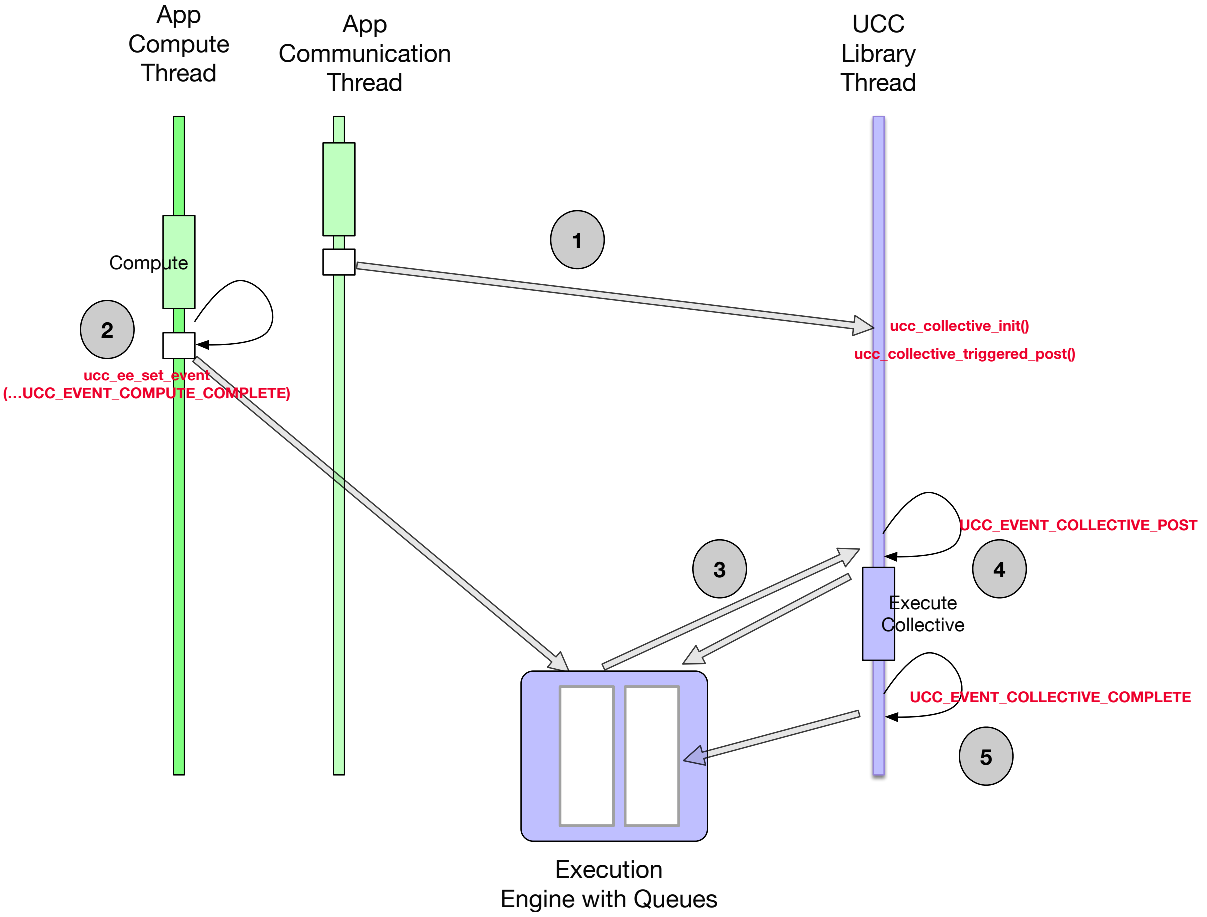

App Compute Thread
App Communication Thread
UCC Library Thread
1
Compute
2
ucc_ee_set_event
(...UCC_EVENT_COMPUTE_COMPLETE)
ucc_collective_init()
ucc_collective_triggered_post()
UCC_EVENT_COLLECTIVE_POST
3
4
Execute Collective
UCC_EVENT_COLLECTIVE_COMPLETE
5
Execution Engine with Queues

Figure 7.1: UCC Execution Engine and Events

Chapter 8

Module Documentation

8.1 Library initialization data-structures

Data Structures

- struct ucc_lib_params

  Structure representing the parameters to customize the library. *More...*
- struct ucc_lib_attr

  Structure representing the attributes of the library. *More...*

Typedefs

- typedef struct ucc_lib_params ucc_lib_params_t

  Structure representing the parameters to customize the library.
- typedef struct ucc_lib_attr ucc_lib_attr_t

  Structure representing the attributes of the library.
- typedef struct ucc_lib_info ∗ ucc_lib_h

  UCC library handle.
- typedef struct ucc_lib_config ∗ ucc_lib_config_h

  UCC library configuration handle.

Enumerations

- enum ucc_coll_type_t {
  UCC_COLL_TYPE_ALLGATHER = UCC_BIT(0),
  UCC_COLL_TYPE_ALLGATHERV = UCC_BIT(1),
  UCC_COLL_TYPE_ALLREDUCE = UCC_BIT(2),
  UCC_COLL_TYPE_ALLTOALL = UCC_BIT(3),
  UCC_COLL_TYPE_ALLTOALLV = UCC_BIT(4),
  UCC_COLL_TYPE_BARRIER = UCC_BIT(5),
  UCC_COLL_TYPE_BCAST = UCC_BIT(6),
  UCC_COLL_TYPE_FANIN = UCC_BIT(7),
  UCC_COLL_TYPE_FANOUT = UCC_BIT(8),
  UCC_COLL_TYPE_GATHER = UCC_BIT(9),
  UCC_COLL_TYPE_GATHERV = UCC_BIT(10),
  UCC_COLL_TYPE_REDUCE = UCC_BIT(11),
  UCC_COLL_TYPE_REDUCE_SCATTER = UCC_BIT(12),
  UCC_COLL_TYPE_REDUCE_SCATTERV = UCC_BIT(13),
  UCC_COLL_TYPE_SCATTER = UCC_BIT(14),
  UCC_COLL_TYPE_SCATTERV = UCC_BIT(15),

UCC_COLL_TYPE_LAST }

Enumeration representing the collective operations.

- enum ucc_reduction_op_t {
UCC_OP_SUM,
UCC_OP_PROD,
UCC_OP_MAX,
UCC_OP_MIN,
UCC_OP_LAND,
UCC_OP_LOR,
UCC_OP_LXOR,
UCC_OP_BAND,
UCC_OP_BOR,
UCC_OP_BXOR,
UCC_OP_MAXLOC,
UCC_OP_MINLOC,
UCC_OP_AVG,
UCC_OP_LAST }

  Enumeration representing the UCC reduction operations.
- enum ucc_thread_mode_t {
UCC_THREAD_SINGLE = 0,
UCC_THREAD_FUNNELED = 1,
UCC_THREAD_MULTIPLE = 2 }

  Enumeration representing the UCC library's thread model.
- enum ucc_coll_sync_type_t {
UCC_NO_SYNC_COLLECTIVES = 0,
UCC_SYNC_COLLECTIVES = 1 }

  Enumeration representing the collective synchronization model.
- enum ucc_lib_params_field {
UCC_LIB_PARAM_FIELD_THREAD_MODE = UCC_BIT(0),
UCC_LIB_PARAM_FIELD_COLL_TYPES = UCC_BIT(1),
UCC_LIB_PARAM_FIELD_REDUCTION_TYPES = UCC_BIT(2),
UCC_LIB_PARAM_FIELD_SYNC_TYPE = UCC_BIT(3) }

  UCC library initialization parameters.
- enum ucc_lib_attr_field {
UCC_LIB_ATTR_FIELD_THREAD_MODE = UCC_BIT(0),
UCC_LIB_ATTR_FIELD_COLL_TYPES = UCC_BIT(1),
UCC_LIB_ATTR_FIELD_REDUCTION_TYPES = UCC_BIT(2),
UCC_LIB_ATTR_FIELD_SYNC_TYPE = UCC_BIT(3) }

8.1.1 Detailed Description

Unified Collective Communications (UCC) Library Specification

UCC is a collective communication operations API and library that is flexible, complete, and feature-rich for current and emerging programming models and runtimes.

Library initialization parameters and data-structures

8.1.2 Data Structure Documentation

8.1.2.1 struct ucc_lib_params

Description

ucc_lib_params_t defines the parameters that can be used to customize the library. The bits in "mask" bit array is defined by ucc_lib_params_field, which correspond to fields in structure 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.

Data Fields

uint64_t	mask	
ucc_thread_mode_t	thread_mode	
uint64_t	coll_types	
uint64_t	reduction_types	
ucc_coll_sync_type_t	sync_type	

8.1.2.2 struct ucc_lib_attr

Description

ucc_lib_attr_t defines the attributes of the library. The bits in "mask" bit array is defined by ucc_lib_attr_field, which correspond to fields in structure ucc_lib_attr_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.

Data Fields

uint64_t	mask	
ucc_thread_mode_t	thread_mode	
uint64_t	coll_types	
uint64_t	reduction_types	
ucc_coll_sync_type_t	sync_type	

8.1.3 Typedef Documentation

8.1.3.1 typedef struct **ucc_lib_params ucc_lib_params_t**

Description

ucc_lib_params_t defines the parameters that can be used to customize the library. The bits in "mask" bit array is defined by ucc_lib_params_field, which correspond to fields in structure 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.

8.1.3.2 typedef struct **ucc_lib_attr ucc_lib_attr_t**

Description

ucc_lib_attr_t defines the attributes of the library. The bits in "mask" bit array is defined by ucc_lib_attr_field, which correspond to fields in structure ucc_lib_attr_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.

8.1.3.3 typedef struct ucc_lib_info∗ **ucc_lib_h**

The ucc library handle is an opaque handle created by the library. It abstracts the collective library. It holds the global information and resources associated with the library. The library handle cannot be passed from one library instance to another.

8.1.3.4 typedef struct ucc_lib_config∗ **ucc_lib_config_h**

8.1.4 Enumeration Type Documentation

8.1.4.1 enum **ucc_coll_type_t**

Library initialization and finalize

Description

ucc_coll_type_t represents the collective operations supported by the UCC library. The exact set of supported collective operations depends on UCC build flags, runtime configuration and available communication transports.

Enumerator

UCC_COLL_TYPE_ALLGATHER

UCC_COLL_TYPE_ALLGATHERV

UCC_COLL_TYPE_ALLREDUCE

UCC_COLL_TYPE_ALLTOALL

UCC_COLL_TYPE_ALLTOALLV

UCC_COLL_TYPE_BARRIER

UCC_COLL_TYPE_BCAST

UCC_COLL_TYPE_FANIN

UCC_COLL_TYPE_FANOUT

UCC_COLL_TYPE_GATHER

UCC_COLL_TYPE_GATHERV

UCC_COLL_TYPE_REDUCE

UCC_COLL_TYPE_REDUCE_SCATTER

UCC_COLL_TYPE_REDUCE_SCATTERV

UCC_COLL_TYPE_SCATTER

UCC_COLL_TYPE_SCATTERV

UCC_COLL_TYPE_LAST

8.1.4.2 enum **ucc_reduction_op_t**

Description

ucc_reduction_op_t represents the UCC reduction operations. It is used by the library initialization routine ucc_init to request the operations expected by the user. It is used by the ucc_lib_attr_t to communicate the operations supported by the library.

Enumerator

UCC_OP_SUM

UCC_OP_PROD

UCC_OP_MAX

UCC_OP_MIN

UCC_OP_LAND

UCC_OP_LOR

UCC_OP_LXOR

UCC_OP_BAND

UCC_OP_BOR

UCC_OP_BXOR

UCC_OP_MAXLOC
UCC_OP_MINLOC
UCC_OP_AVG
UCC_OP_LAST

8.1.4.3 enum **ucc_thread_mode_t**

Description

ucc_thread_mode_t is used to initialize the UCC library's thread mode. The UCC library can be configured in three thread modes UCC_THREAD_SINGLE, UCC_THREAD_FUNNELED, and UCC_THREAD_M↵ ULTIPLE. In the UCC_THREAD_SINGLE mode, the user program must not be multithreaded. In the UCC_THREAD_FUNNELED mode, the user program may be multithreaded. However, all UCC interfaces should be invoked from the same thread. In the UCC_THREAD_MULTIPLE mode, the user program can be multithreaded and any thread may invoke the UCC operations.

Enumerator

UCC_THREAD_SINGLE Single-threaded library model
UCC_THREAD_FUNNELED Funnel thread model
UCC_THREAD_MULTIPLE Multithread library model

8.1.4.4 enum **ucc_coll_sync_type_t**

Description

ucc_coll_sync_type_t represents the collective synchronization models. Currently, it supports two synchronization models synchronous and non-synchronous collective models. In the synchronous collective model, the collective communication is not started until participants have not entered the collective operation, and it is not completed until all participants have not completed the collective. In the non-synchronous collective model, collective communication can be started as soon as the participant enters the collective operation and is completed as soon as it completes locally.

Enumerator

UCC_NO_SYNC_COLLECTIVES Non-synchronous collectives
UCC_SYNC_COLLECTIVES Synchronous collectives

8.1.4.5 enum **ucc_lib_params_field**

Enumerator

UCC_LIB_PARAM_FIELD_THREAD_MODE
UCC_LIB_PARAM_FIELD_COLL_TYPES
UCC_LIB_PARAM_FIELD_REDUCTION_TYPES
UCC_LIB_PARAM_FIELD_SYNC_TYPE

8.1.4.6 enum **ucc_lib_attr_field**

Enumerator

UCC_LIB_ATTR_FIELD_THREAD_MODE
UCC_LIB_ATTR_FIELD_COLL_TYPES
UCC_LIB_ATTR_FIELD_REDUCTION_TYPES
UCC_LIB_ATTR_FIELD_SYNC_TYPE

8.2 Datatypes data-structures and functions

Data Structures

- struct ucc_reduce_cb_params

  Descriptor of user-defined reduction callback. More...
- struct ucc_generic_dt_ops

  UCC generic data type descriptor.
- struct ucc_generic_dt_ops.reduce

  User-defined reduction callback.

Typedefs

- typedef uint64_t ucc_datatype_t

  Enumeration representing the UCC library's datatype.
- typedef struct ucc_reduce_cb_params ucc_reduce_cb_params_t

  Descriptor of user-defined reduction callback.
- typedef struct ucc_generic_dt_ops ucc_generic_dt_ops_t

  UCC generic data type descriptor.

Enumerations

- enum ucc_generic_dt_ops_field { UCC_GENERIC_DT_OPS_FIELD_FLAGS = UCC_BIT(0) }
- enum ucc_generic_dt_ops_flags_t {
  UCC_GENERIC_DT_OPS_FLAG_CONTIG = UCC_BIT(0),
  UCC_GENERIC_DT_OPS_FLAG_REDUCE = UCC_BIT(1) }

  Flags that can be specified for generic datatype.
- enum ucc_dt_type_t {
  UCC_DATATYPE_PREDEFINED = 0,
  UCC_DATATYPE_GENERIC = UCC_BIT(0),
  UCC_DATATYPE_SHIFT = 3,
  UCC_DATATYPE_CLASS_MASK = UCC_MASK(UCC_DATATYPE_SHIFT) }

  Helper enum for generic/predefined datatype representation.

Functions

- ucc_status_t ucc_dt_create_generic (const ucc_generic_dt_ops_t ∗ops, void ∗context, ucc_datatype_t ∗datatype_p)

  Create a generic datatype.
- void ucc_dt_destroy (ucc_datatype_t datatype)

  Destroy generic datatype.

Variables

- void ∗(∗ ucc_generic_dt_ops::start_pack)(void ∗context, const void ∗buffer, size_t count)

  Start a packing request.
- void ∗(∗ ucc_generic_dt_ops::start_unpack)(void ∗context, void ∗buffer, size_t count)

  Start an unpacking request.
- size_t(∗ ucc_generic_dt_ops::packed_size)(void ∗state)

  Get the total size of packed data.
- size_t(∗ ucc_generic_dt_ops::pack)(void ∗state, size_t offset, void ∗dest, size_t max_length)

Pack data.

- ucc_status_t(∗ ucc_generic_dt_ops::unpack)(void ∗state, size_t offset, const void ∗src, size_t length)

  Unpack data.
- void(∗ ucc_generic_dt_ops::finish)(void ∗state)

  Finish packing/unpacking.
- struct {
  ucc_status_t(∗ cb)(const ucc_reduce_cb_params_t ∗params)
  void ∗ cb_ctx
  } ucc_generic_dt_ops::reduce

  User-defined reduction callback.

8.2.1 Detailed Description

Datatypes data-structures and functions

8.2.2 Data Structure Documentation

8.2.2.1 struct ucc_reduce_cb_params

This structure is the argument to the reduce.cb callback. It must implement the reduction of n_vectors + 1 data vectors each containing "count" elements. First vector is "src1", other n_vectors have start address v_j = src2 + count ∗ dt_extent ∗ stride ∗ j. The result is stored in dst, so that dst[i] = src1[i] + v0[i] + v1[i] + ... +v_nvectors[i], for i in [0:count), where "+" represents user-defined reduction of 2 elements

Data Fields

uint64_t	mask	
void ∗	src1	
void ∗	src2	
void ∗	dst	
size_t	n_vectors	
size_t	count	
size_t	stride	
ucc_dt_generic_t ∗	dt	
void ∗	cb_ctx	

8.2.3 Typedef Documentation

8.2.3.1 typedef uint64_t **ucc_datatype_t**

Description

ucc_datatype_t represents the datatypes supported by the UCC library's collective and reduction operations. The predefined operations are signed and unsigned integers of various sizes, float 16, 32, and 64, and user-defined datatypes. User-defined datatypes are created using ucc_dt_create_generic interface and can support user-defined reduction operations. Predefined reduction operations can be used only with predefined datatypes.

8.2.3.2 typedef struct **ucc_reduce_cb_params ucc_reduce_cb_params_t**

This structure is the argument to the reduce.cb callback. It must implement the reduction of n_vectors + 1 data vectors each containing "count" elements. First vector is "src1", other n_vectors have start address v_j

= src2 + count ∗ dt_extent ∗ stride ∗ j. The result is stored in dst, so that dst[i] = src1[i] + v0[i] + v1[i] + ... +v_nvectors[i], for i in [0:count), where "+" represents user-defined reduction of 2 elements

8.2.3.3 typedef struct **ucc_generic_dt_ops ucc_generic_dt_ops_t**

This structure provides a generic datatype descriptor that is used to create user-defined datatypes.

8.2.4 Enumeration Type Documentation

8.2.4.1 enum **ucc_generic_dt_ops_field**

Enumerator

UCC_GENERIC_DT_OPS_FIELD_FLAGS

8.2.4.2 enum **ucc_generic_dt_ops_flags_t**

Enumerator

UCC_GENERIC_DT_OPS_FLAG_CONTIG If set, the created datatype represents a contiguous memory region with the size specified in ucc_generic_dt_ops::contig_size field of ucc_generic_dt_ops

UCC_GENERIC_DT_OPS_FLAG_REDUCE If set, the created datatype has user-defined reduction operation associated with it. reduce.cb and reduce.ctx fields of ucc_generic_dt_ops must be initialized. Collective operations that involve reduction (allreduce, reduce, reduce_scatter/v) can use user-defined data-types only when this flag is set.

8.2.4.3 enum **ucc_dt_type_t**

Enumerator

UCC_DATATYPE_PREDEFINED

UCC_DATATYPE_GENERIC

UCC_DATATYPE_SHIFT

UCC_DATATYPE_CLASS_MASK

8.2.5 Function Documentation

8.2.5.1 **ucc_status_t** ucc_dt_create_generic (const **ucc_generic_dt_ops_t** ∗ ops, void ∗ context, **ucc_datatype_t** ∗ datatype_p)

This routine creates a generic datatype object. The generic datatype is described by the *ops* object which provides a table of routines defining the operations for generic datatype manipulation. Typically, generic datatypes are used for integration with datatype engines provided with MPI implementations (MPICH, Open MPI, etc). The application is responsible for releasing the *datatype_p* object using ucc_dt_destroy() routine.

Parameters

in	*ops*	Generic datatype function table as defined by ucc_generic_dt_ops_t .
in	*context*	Application defined context passed to this routine. The context is passed as a parameter to the routines in the *ops* table.

out	*datatype_p*	A pointer to datatype object.

Returns

Error code as defined by ucc_status_t

8.2.5.2 void ucc_dt_destroy (**ucc_datatype_t** datatype)

8.2.6 Variable Documentation

8.2.6.1 void∗(∗ ucc_generic_dt_ops::start_pack) (void ∗context, const void ∗buffer, size_t count)

The pointer refers to application defined start-to-pack routine.

Parameters

in	*context*	User-defined context.
in	*buffer*	Buffer to pack.
in	*count*	Number of elements to pack into the buffer.

Returns

A custom state that is passed to the subsequent pack() routine.

8.2.6.2 void∗(∗ ucc_generic_dt_ops::start_unpack) (void ∗context, void ∗buffer, size_t count)

The pointer refers to application defined start-to-unpack routine.

Parameters

in	*context*	User-defined context.
in	*buffer*	Buffer to unpack to.
in	*count*	Number of elements to unpack in the buffer.

Returns

A custom state that is passed later to the subsequent unpack() routine.

8.2.6.3 size_t(∗ ucc_generic_dt_ops::packed_size) (void ∗state)

The pointer refers to user defined routine that returns the size of data in a packed format.

Parameters

in	*state*	State as returned by start_pack() routine.

Returns

The size of the data in a packed form.

8.2.6.4 size_t(∗ ucc_generic_dt_ops::pack) (void ∗state, size_t offset, void ∗dest, size_t max_length)

The pointer refers to application defined pack routine.

Parameters

in	*state*	State as returned by start_pack() routine.
in	*offset*	Virtual offset in the output stream.
in	*dest*	Destination buffer to pack the data.
in	*max_length*	Maximum length to pack.

Returns

The size of the data that was written to the destination buffer. Must be less than or equal to *max_length*.

8.2.6.5 **ucc_status_t**(∗ ucc_generic_dt_ops::unpack) (void ∗state, size_t offset, const void ∗src, size_t length)

The pointer refers to application defined unpack routine.

Parameters

in	*state*	State as returned by start_unpack() routine.
in	*offset*	Virtual offset in the input stream.
in	*src*	Source to unpack the data from.
in	*length*	Length to unpack.

Returns

UCC_OK or an error if unpacking failed.

8.2.6.6 void(∗ ucc_generic_dt_ops::finish) (void ∗state)

The pointer refers to application defined finish routine.

Parameters

in	*state*	State as returned by start_pack() and start_unpack() routines.

8.2.6.7 struct { ... } ucc_generic_dt_ops::reduce

The pointer refers to user-defined reduction routine.

Parameters

in	*params*	reduction descriptor

8.2.6.8 ucc_status_t(∗ { ... } ::cb) (const **ucc_reduce_cb_params_t** ∗params)

8.2.6.9 void∗ { ... } ::cb_ctx

8.3 Library initialization and finalization routines

Functions

- ucc_status_t ucc_lib_config_read (const char *env_prefix, const char *filename, ucc_lib_config_h *config)

  The ucc_lib_config_read routine provides a method to read library configuration from the environment and create configuration descriptor.
- void ucc_lib_config_release (ucc_lib_config_h config)

  The ucc_lib_config_release routine releases the configuration descriptor.
- void ucc_lib_config_print (const ucc_lib_config_h config, FILE *stream, const char *title, ucc_↩ config_print_flags_t print_flags)

  The ucc_lib_config_print routine prints the configuration information.
- ucc_status_t ucc_lib_config_modify (ucc_lib_config_h config, const char *name, const char *value)

  The ucc_lib_config_modify routine modifies the runtime configuration as described by the descriptor.
- void ucc_get_version (unsigned *major_version, unsigned *minor_version, unsigned *release_number)

  Get UCC library version.
- const char * ucc_get_version_string (void)

  Get UCC library version as a string.
- static ucc_status_t ucc_init (const ucc_lib_params_t *params, const ucc_lib_config_h config, ucc↩ _lib_h *lib_p)

  The ucc_init initializes the UCC library.
- ucc_status_t ucc_finalize (ucc_lib_h lib_p)

  The ucc_finalize routine finalizes the UCC library.
- ucc_status_t ucc_lib_get_attr (ucc_lib_h lib_p, ucc_lib_attr_t *lib_attr)

  The ucc_lib_get_attr routine queries the library attributes.

8.3.1 Detailed Description

Library initialization and finalization routines

8.3.2 Function Documentation

8.3.2.1 **ucc_status_t** ucc_lib_config_read (const char * env_prefix, const char * filename, **ucc_lib_config_h** * config)

Parameters

out	*env_prefix*	If not NULL, the routine searches for the environment variables with the prefix UCC_<env_prefix>. Otherwise, the routines search for the environment variables that start with the prefix @ UCC_.
in	*filename*	If not NULL, read configuration values from the file defined by *filename*. If the file does not exist, it will be ignored and no error will be reported to the user.
out	*config*	Pointer to configuration descriptor as defined by ucc_lib_config_h.

Description

ucc_lib_config_read allocates the ucc_lib_config_h handle and fetches the configuration values from the run-time environment. The run-time environment supported are environment variables or a configuration file.

Returns

Error code as defined by ucc_status_t

8.3.2.2 void ucc_lib_config_release (**ucc_lib_config_h** config)

Parameters

in	*config*	Pointer to the configuration descriptor to be released. Configuration descriptor as defined by ucc_lib_config_h.

Description

The routine releases the configuration descriptor that was allocated through

ucc_lib_config_read() routine.

8.3.2.3 void ucc_lib_config_print (const **ucc_lib_config_h** config, FILE * stream, const char * title, **ucc_config_print_flags_t** print_flags)

Parameters

in	*config*	ucc_lib_config_h "Configuration descriptor" to print.
in	*stream*	Output stream to print the configuration to.
in	*title*	Configuration title to print.
in	*print_flags*	Flags that control various printing options.

Description

The routine prints the configuration information that is stored in ucc_lib_config_h "configuration" descriptor.

8.3.2.4 **ucc_status_t** ucc_lib_config_modify (**ucc_lib_config_h** config, const char * name, const char * value)

Parameters

in	*config*	Pointer to the configuration descriptor to be modified
in	*name*	Configuration variable to be modified
in	*value*	Configuration value to set

Description

The ucc_lib_config_modify routine sets the value of identifier "name" to "value".

Returns

Error code as defined by ucc_status_t

8.3.2.5 void ucc_get_version (unsigned * major_version, unsigned * minor_version, unsigned * release_number)

This routine returns the UCC library version.

Parameters

out	*major_version*	Filled with library major version.
out	*minor_version*	Filled with library minor version.
out	*release_number*	Filled with library release number.

8.3.2.6 const char* ucc_get_version_string (void)

This routine returns the UCC library version as a string which consists of: "major.minor.release".

8.3.2.7 static **ucc_status_t** ucc_init (const **ucc_lib_params_t** * params, const **ucc_lib_config_h** config, **ucc_lib_h** * lib_p) [inline], [static]

Parameters

in	*params*	User provided parameters to customize the library functionality
in	*config*	UCC configuration descriptor allocated through ucc_config_read() routine.
out	*lib_p*	UCC library handle

Description

A local operation to initialize and allocate the resources for the UCC operations. The parameters passed using the ucc_lib_params_t and 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 ucc_status_t is returned.

Returns

Error code as defined by ucc_status_t

8.3.2.8 **ucc_status_t** ucc_finalize (**ucc_lib_h** lib_p)

Parameters

in	*lib_p*	Handle to ucc_lib_h "UCC library".

Description

A local operation to release the resources and cleanup. All participants that invoked ucc_init should call this routine.

Returns

Error code as defined by ucc_status_t

8.3.2.9 **ucc_status_t** ucc_lib_get_attr (**ucc_lib_h** lib_p, **ucc_lib_attr_t** * lib_attr)

Parameters

out	*lib_attr*	Library attributes
in	*lib_p*	Input library object

Description

A query operation to get the attributes of the library object. The attributes are library configured values and reflect the choices made by the library implementation.

Returns

Error code as defined by ucc_status_t

8.4 Internal library routines

Functions

- ucc_status_t ucc_init_version (unsigned api_major_version, unsigned api_minor_version, const ucc_lib_params_t *params, const ucc_lib_config_h config, ucc_lib_h *lib_p)

  The ucc_init_version is an internal routine that checks compatibility with a particular UCC API version. ucc_init should be used to create the UCC library handle.

8.4.1 Detailed Description

Internal library routines

8.4.2 Function Documentation

8.4.2.1 **ucc_status_t** ucc_init_version (unsigned api_major_version, unsigned api_minor_version, const **ucc_lib_params_t** * params, const **ucc_lib_config_h** config, **ucc_lib_h** * lib_p)

8.5 Context abstraction data-structures

Data Structures

- struct ucc_oob_coll

  OOB collective operation for creating the context.
- struct ucc_mem_map
- struct ucc_mem_map_params
- struct ucc_context_params

  Structure representing the parameters to customize the context. More...
- struct ucc_context_attr

  Structure representing context attributes. More...

Typedefs

- typedef struct ucc_oob_coll ucc_oob_coll_t

  OOB collective operation for creating the context.
- typedef struct ucc_mem_map ucc_mem_map_t
- typedef struct ucc_mem_map_params ucc_mem_map_params_t
- typedef struct ucc_context_params ucc_context_params_t

  Structure representing the parameters to customize the context.
- typedef struct ucc_context_attr ucc_context_attr_t

  Structure representing context attributes.
- typedef struct ucc_context ∗ ucc_context_h

  UCC context.
- typedef struct ucc_context_config ∗ ucc_context_config_h

  UCC context configuration handle.

Enumerations

- enum ucc_context_type_t {
  UCC_CONTEXT_EXCLUSIVE = 0,
  UCC_CONTEXT_SHARED }
- enum ucc_context_params_field {
  UCC_CONTEXT_PARAM_FIELD_TYPE = UCC_BIT(0),
  UCC_CONTEXT_PARAM_FIELD_SYNC_TYPE = UCC_BIT(1),
  UCC_CONTEXT_PARAM_FIELD_OOB = UCC_BIT(2),
  UCC_CONTEXT_PARAM_FIELD_ID = UCC_BIT(3),
  UCC_CONTEXT_PARAM_FIELD_MEM_PARAMS = UCC_BIT(4) }
- enum ucc_context_attr_field {
  UCC_CONTEXT_ATTR_FIELD_TYPE = UCC_BIT(0),
  UCC_CONTEXT_ATTR_FIELD_SYNC_TYPE = UCC_BIT(1),
  UCC_CONTEXT_ATTR_FIELD_CTX_ADDR = UCC_BIT(2),
  UCC_CONTEXT_ATTR_FIELD_CTX_ADDR_LEN = UCC_BIT(3),
  UCC_CONTEXT_ATTR_FIELD_WORK_BUFFER_SIZE = UCC_BIT(4) }

8.5.1 Detailed Description

Data-structures associated with context creation and management routines

8.5.2 Data Structure Documentation

8.5.2.1 struct ucc_mem_map

Data Fields

void *	address	the address of a buffer to be attached to a UCC context
size_t	len	the length of the buffer

8.5.2.2 struct ucc_mem_map_params

Data Fields

ucc_mem_map_t *	segments	array of ucc_mem_map elements
uint64_t	n_segments	the number of ucc_mem_map elements

8.5.2.3 struct ucc_context_params

Description

ucc_context_params_t defines the parameters that can be used to customize the context. The "mask" bit array fields are defined by ucc_context_params_field. The bits in "mask" bit array is defined by ucc_context_params_field, which correspond to fields in structure ucc_context_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.

Data Fields

uint64_t	mask	
ucc_context_type_t	type	
ucc_coll_sync_type_t	sync_type	
ucc_context_oob_coll_t	oob	
uint64_t	ctx_id	
ucc_mem_map_params_t	mem_params	

8.5.2.4 struct ucc_context_attr

Description

ucc_context_attr_t defines the attributes of the context. The bits in "mask" bit array is defined by ucc_context_attr_field, which correspond to fields in structure ucc_context_attr_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.

Data Fields

uint64_t	mask	
ucc_context_type_t	type	
ucc_coll_sync_type_t	sync_type	

ucc_context_addr_h	ctx_addr	
ucc_context_addr_len_t	ctx_addr_len	
uint64_t	global_work_buffer_size	

8.5.3 Typedef Documentation

8.5.3.1 typedef struct **ucc_oob_coll ucc_oob_coll_t**

8.5.3.2 typedef struct **ucc_mem_map ucc_mem_map_t**

8.5.3.3 typedef struct **ucc_mem_map_params ucc_mem_map_params_t**

8.5.3.4 typedef struct **ucc_context_params ucc_context_params_t**

Description

ucc_context_params_t defines the parameters that can be used to customize the context. The "mask" bit array fields are defined by ucc_context_params_field. The bits in "mask" bit array is defined by ucc_context_params_field, which correspond to fields in structure ucc_context_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.

8.5.3.5 typedef struct **ucc_context_attr ucc_context_attr_t**

Description

ucc_context_attr_t defines the attributes of the context. The bits in "mask" bit array is defined by ucc_context_attr_field, which correspond to fields in structure ucc_context_attr_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.

8.5.3.6 typedef struct ucc_context∗ **ucc_context_h**

The UCC context is an opaque handle to abstract the network resources for collective operations. The network resources could be either software or hardware. Based on the type of the context, the resources can be shared or either be exclusively used. The UCC context is required but not sufficient to execute a collective operation.

8.5.3.7 typedef struct ucc_context_config∗ **ucc_context_config_h**

8.5.4 Enumeration Type Documentation

8.5.4.1 enum **ucc_context_type_t**

Enumerator

UCC_CONTEXT_EXCLUSIVE

UCC_CONTEXT_SHARED

8.5.4.2 enum **ucc_context_params_field**

Enumerator

UCC_CONTEXT_PARAM_FIELD_TYPE
UCC_CONTEXT_PARAM_FIELD_SYNC_TYPE
UCC_CONTEXT_PARAM_FIELD_OOB
UCC_CONTEXT_PARAM_FIELD_ID
UCC_CONTEXT_PARAM_FIELD_MEM_PARAMS

8.5.4.3 enum **ucc_context_attr_field**

Enumerator

UCC_CONTEXT_ATTR_FIELD_TYPE
UCC_CONTEXT_ATTR_FIELD_SYNC_TYPE
UCC_CONTEXT_ATTR_FIELD_CTX_ADDR
UCC_CONTEXT_ATTR_FIELD_CTX_ADDR_LEN
UCC_CONTEXT_ATTR_FIELD_WORK_BUFFER_SIZE

8.6 Context abstraction routines

Functions

- ucc_status_t ucc_context_config_read (ucc_lib_h lib_handle, const char *filename, ucc_context_config_h *config)

  Routine reads the configuration information for contexts from the runtime enviornment and creates the configuration descriptor.
- void ucc_context_config_release (ucc_context_config_h config)

  The ucc_context_config_release routine releases the configuration descriptor.
- void ucc_context_config_print (const ucc_context_config_h config, FILE *stream, const char *title, ucc_config_print_flags_t print_flags)

  The ucc_context_config_print routine prints the configuration information.
- ucc_status_t ucc_context_config_modify (ucc_context_config_h config, const char *component, const char *name, const char *value)

  The ucc_context_config_modify routine modifies the runtime configuration of UCC context (optionally for a given CLS)
- 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)

  The ucc_context_create routine creates the context handle.
- ucc_status_t ucc_context_progress (ucc_context_h context)

  The ucc_context_progress routine progresses the operations on the context handle.
- ucc_status_t ucc_context_destroy (ucc_context_h context)

  The ucc_context_destroy routine frees the context handle.
- ucc_status_t ucc_context_get_attr (ucc_context_h context, ucc_context_attr_t *context_attr)

  The routine queries the attributes of the context handle.

8.6.1 Detailed Description

Context create and management routines

8.6.2 Function Documentation

8.6.2.1 **ucc_status_t** ucc_context_config_read (**ucc_lib_h** lib_handle, const char * filename, **ucc_context_config_h** * config)

Parameters

in	*lib_handle*	Library handle
in	*filename*	If not NULL, read configuration values from the file defined by *filename*. If the file does not exist, it will be ignored and no error will be reported to the user.
out	*config*	Pointer to configuration descriptor as defined by ucc_context_config_h.

Description

ucc_context_config_read allocates the ucc_lib_config_h handle and fetches the configuration values from the run-time environment. The run-time environment supported are environment variables or a configuration file. It uses the env_prefix from ucc_lib_config_read. If env_prefix is not NULL, the routine searches for the environment variables with the prefix UCC_<env_prefix>. Otherwise, the routines search for the environment variables that start with the prefix @ UCC_.

Returns

Error code as defined by ucc_status_t

8.6.2.2 void ucc_context_config_release (**ucc_context_config_h** config)

Parameters

in	*config*	Pointer to the configuration descriptor to be released. Configuration descriptor as defined by ucc_context_config_h

Description

The routine releases the configuration descriptor that was allocated through ucc_context_config_read() routine.

8.6.2.3 void ucc_context_config_print (const **ucc_context_config_h** config, FILE ∗ stream, const char ∗ title, **ucc_config_print_flags_t** print_flags)

Parameters

in	*config*	ucc_context_config_h "Configuration descriptor" to print.
in	*stream*	Output stream to print the configuration to.
in	*title*	Configuration title to print.
in	*print_flags*	Flags that control various printing options.

Description

The routine prints the configuration information that is stored in ucc_context_config_h "configuration" descriptor.

8.6.2.4 **ucc_status_t** ucc_context_config_modify (**ucc_context_config_h** config, const char ∗ component, const char ∗ name, const char ∗ value)

Parameters

in	*config*	Pointer to the configuration descriptor to be modified
in	*component*	CL/TL component (e.g. "tl/ucp" or "cl/basic") or NULL. If NULL then core context config is modified.
in	*name*	Configuration variable to be modified
in	*value*	Configuration value to set

Description

The ucc_context_config_modify routine sets the value of identifier "name" to "value" for a specified CL.

Returns

Error code as defined by ucc_status_t

8.6.2.5 **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)

Parameters

in	*lib_handle*	Library handle
in	*params*	Customizations for the communication context
in	*config*	Configuration for the communication context to read from environment
out	*context*	Pointer to the newly created communication context

Description

The ucc_context_create creates the context and 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 context object will not be created and corresponding error code as defined by ucc_status_t is returned.

Returns

Error code as defined by ucc_status_t

8.6.2.6 **ucc_status_t** ucc_context_progress (**ucc_context_h** context)

Parameters

in	*context*	Communication context handle to be progressed

Description

The ucc_context_progress routine progresses the operations on the content handle. It does not block for lack of resources or communication.

Returns

Error code as defined by ucc_status_t

8.6.2.7 **ucc_status_t** ucc_context_destroy (**ucc_context_h** context)

Parameters

in	*context*	Communication context handle to be released

Description

ucc_context_destroy routine releases the resources associated with the handle *context*. All teams associated with the team should be released before this. It is invalid to associate any team with this handle after the routine is called.

Returns

Error code as defined by ucc_status_t

8.6.2.8 **ucc_status_t** ucc_context_get_attr (**ucc_context_h** context, **ucc_context_attr_t** * context_attr)

Parameters

in	*context*	Communication context
out	*context_attr*	Attributes of the communication context

Description

ucc_context_get_attr routine queries the context handle attributes described by ucc_context_attr.

Returns

Error code as defined by ucc_status_t

8.7 Team abstraction data-structures

Data Structures

- struct ucc_team_p2p_conn
- struct ucc_ep_map_strided
- struct ucc_ep_map_array
- struct ucc_ep_map_cb
- struct ucc_ep_map_t
- union ucc_ep_map_t.__unnamed__
- struct ucc_team_params

  Structure representing the parameters to customize the team. More...
- struct ucc_team_attr

  Structure representing the team attributes. More...

Typedefs

- typedef struct ucc_team_p2p_conn ucc_team_p2p_conn_t
- typedef struct ucc_ep_map_t ucc_ep_map_t
- typedef struct ucc_team_params ucc_team_params_t

  Structure representing the parameters to customize the team.
- typedef struct ucc_team_attr ucc_team_attr_t

  Structure representing the team attributes.
- typedef struct ucc_team ∗ ucc_team_h

  UCC team handle.
- typedef void ∗ ucc_p2p_conn_t
- typedef void ∗ ucc_context_addr_h
- typedef size_t ucc_context_addr_len_t

Enumerations

- enum ucc_team_params_field {
  UCC_TEAM_PARAM_FIELD_ORDERING = UCC_BIT(0),
  UCC_TEAM_PARAM_FIELD_OUTSTANDING_COLLS = UCC_BIT(1),
  UCC_TEAM_PARAM_FIELD_EP = UCC_BIT(2),
  UCC_TEAM_PARAM_FIELD_EP_LIST = UCC_BIT(3),
  UCC_TEAM_PARAM_FIELD_EP_RANGE = UCC_BIT(4),
  UCC_TEAM_PARAM_FIELD_TEAM_SIZE = UCC_BIT(5),
  UCC_TEAM_PARAM_FIELD_SYNC_TYPE = UCC_BIT(6),
  UCC_TEAM_PARAM_FIELD_OOB = UCC_BIT(7),
  UCC_TEAM_PARAM_FIELD_P2P_CONN = UCC_BIT(8),
  UCC_TEAM_PARAM_FIELD_MEM_PARAMS = UCC_BIT(9),
  UCC_TEAM_PARAM_FIELD_EP_MAP = UCC_BIT(10),
  UCC_TEAM_PARAM_FIELD_ID = UCC_BIT(11),
  UCC_TEAM_PARAM_FIELD_FLAGS = UCC_BIT(12) }
- enum ucc_team_attr_field {
  UCC_TEAM_ATTR_FIELD_POST_ORDERING = UCC_BIT(0),
  UCC_TEAM_ATTR_FIELD_OUTSTANDING_CALLS = UCC_BIT(1),
  UCC_TEAM_ATTR_FIELD_EP = UCC_BIT(2),
  UCC_TEAM_ATTR_FIELD_EP_RANGE = UCC_BIT(3),
  UCC_TEAM_ATTR_FIELD_SYNC_TYPE = UCC_BIT(4),
  UCC_TEAM_ATTR_FIELD_MEM_PARAMS = UCC_BIT(5),
  UCC_TEAM_ATTR_FIELD_SIZE = UCC_BIT(6),
  UCC_TEAM_ATTR_FIELD_EPS = UCC_BIT(7) }

- enum ucc_team_flags { UCC_TEAM_FLAG_COLL_WORK_BUFFER = UCC_BIT(0) }
- enum ucc_post_ordering_t {
  UCC_COLLECTIVE_POST_ORDERED = 0,
  UCC_COLLECTIVE_POST_UNORDERED = 1,
  UCC_COLLECTIVE_INIT_ORDERED = 2,
  UCC_COLLECTIVE_INIT_UNORDERED = 3,
  UCC_COLLECTIVE_INIT_AND_POST_ORDERED = 4,
  UCC_COLLECTIVE_INIT_AND_POST_UNORDERED = 5 }
- enum ucc_ep_range_type_t {
  UCC_COLLECTIVE_EP_RANGE_CONTIG = 0,
  UCC_COLLECTIVE_EP_RANGE_NONCONTIG = 1 }
- enum ucc_ep_map_type_t {
  UCC_EP_MAP_FULL = 1,
  UCC_EP_MAP_STRIDED = 2,
  UCC_EP_MAP_ARRAY = 3,
  UCC_EP_MAP_CB = 4 }

8.7.1 Detailed Description

Data-structures associated with team create and management routines

8.7.2 Data Structure Documentation

8.7.2.1 struct ucc_ep_map_strided

Data Fields

uint64_t	start	
int64_t	stride	

8.7.2.2 struct ucc_ep_map_array

Data Fields

void *	map	
size_t	elem_size	4 if array is int, 8 if e.g. uint64_t

8.7.2.3 struct ucc_ep_map_t

Data Fields

ucc_ep_map_type_t	type	
uint64_t	ep_num	number of eps mapped to ctx
union ucc_ep_map_t	__unnamed__	

8.7.2.4 union ucc_ep_map_t.__unnamed__

Data Fields

struct ucc_ep_map_strided	strided	
struct ucc_ep_map_array	array	
struct ucc_ep_map_cb	cb	

8.7.2.5 struct ucc_team_params

Description

ucc_team_params_t defines the parameters that can be used to customize the team. The "mask" bit array fields are defined by ucc_team_params_field. The bits in "mask" bit array is defined by ucc_team_params_field, which correspond to fields in structure ucc_team_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.

Data Fields

uint64_t	mask	
uint64_t	flags	
ucc_post_ordering_t	ordering	ucc_team_params::ordering is set to one the values defined by ucc_post_ordering_t
uint64_t	outstanding_colls	ucc_team_params::outstanding_colls represents the number of outstanding non-blocking calls the user expects to post to the team. If the user posts more non-blocking calls than set, the behavior is undefined. If not set, there is no limit on the number of outstanding calls to be posted.
uint64_t	ep	ucc_team_params::ep The endpoint is a non-negative unique integer identifying the participant in the collective. If ep is not set, and ucc_team_params::oob is not set, the library generates the ep. The generated ep can be queried using the ucc_team_get_attr interface.
uint64_t *	ep_list	ucc_team_params::ep_list The endpoint list provides the list of eps participating to create the team.
ucc_ep_range_type_t	ep_range	ucc_team_params::ep_range can be either contiguous or not contiguous. It is a hint to the library.
uint64_t	team_size	ucc_team_params::team_size The team size is the number of participants in the team. If ucc_team_params::oob is provided, the team size and ucc_oob_coll::n_oob_eps should be the same.
ucc_coll_sync_type_t	sync_type	ucc_team_params::sync_type The options for sync_type are provided by ucc_coll_sync_type_t
ucc_team_oob_coll_t	oob	ucc_team_params::oob The signature of the function is defined by ucc_oob_coll_t . The oob is used for exchanging information between the team participants during team creation. The user is responsible for implementing the oob operation. The relation between ucc_team_params::ep and ucc_oob_coll::oob_ep is defined as below: • When both are not provided. The library is responsible for generating the ep, which can be then queried via the ucc_team_get_attr interface. This requires, however, ucc_params_t ep_map to be set and context created by ucc_oob_coll. The behavior is undefined, when neither ucc_team_params::ep or ucc_team_params::ep_map, or ucc_team_params::oob is not set. • When ucc_team_params::ep is provided and ucc_team_params::oob is not provided. The "ep" is the unique integer for the participant. • When ucc_oob_coll::oob_ep is provided and ucc_team_params::ep is not provided. The "ep" will be equivalent to ucc_oob_coll::oob_ep. • When both are provided, the ucc_oob_coll::oob_ep and ucc_team_params_t::ep should be same. Otherwise, it is undefined.
ucc_team_p2p_conn_t	p2p_conn	ucc_team_params::p2p_conn is a callback function for the gathering the point-to-point communication information.

ucc_mem_map_params_t	mem_params	ucc_team_params::mem_params provides an ability to attach a buffer to the team. This can be used as input/output or control buffer for the team. Typically, it can be useful for one-sided collective implementation.
ucc_ep_map_t	ep_map	ucc_team_params::ep_map provides a mapping between ucc_oob_coll::oob_ep used by the team and ucc_oob_coll::oob_ep used by the context. The mapping options are defined by ucc_ep_map_t. The definition is valid only when context is created with an ucc_oob_coll.
uint64_t	id	ucc_team_params::id The team id is a unique integer identifying the team that is active. The integer is unique within the process and not the job .i.e., any two active non-overlapping teams can have the same id. This semantic helps to avoid a global information exchange .i.e, the processes or threads not participating in the particular, need not participate in the team creation. If not provided, the team id is created internally. For the MPI programming model, this can be inherited from the MPI communicator id.

8.7.2.6 struct ucc_team_attr

Description

ucc_team_attr_t defines the attributes of the team. The bits in "mask" bit array is defined by ucc_team_attr_field, which correspond to fields in structure ucc_team_attr_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.

Data Fields

uint64_t	mask	
ucc_post_ordering_t	ordering	
uint64_t	outstanding_colls	
uint64_t	ep	
ucc_ep_range_type_t	ep_range	
ucc_coll_sync_type_t	sync_type	
ucc_mem_map_params_t	mem_params	
uint32_t	size	
uint64_t *	eps	

8.7.3 Typedef Documentation

8.7.3.1 typedef struct **ucc_team_p2p_conn ucc_team_p2p_conn_t**

8.7.3.2 typedef struct **ucc_ep_map_t ucc_ep_map_t**

8.7.3.3 typedef struct **ucc_team_params ucc_team_params_t**

Description

ucc_team_params_t defines the parameters that can be used to customize the team. The "mask" bit array fields are defined by ucc_team_params_field. The bits in "mask" bit array is defined by ucc_team_ params_field, which correspond to fields in structure ucc_team_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.

8.7.3.4 typedef struct **ucc_team_attr** **ucc_team_attr_t**

Description

ucc_team_attr_t defines the attributes of the team. The bits in "mask" bit array is defined by ucc_team_attr_field, which correspond to fields in structure ucc_team_attr_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.

8.7.3.5 typedef struct ucc_team∗ **ucc_team_h**

The UCC team handle is an opaque handle created by the library. It abstracts the group resources required for the collective operations and participants of the collective operation. The participants of the collective operation can be an OS process or thread.

8.7.3.6 typedef void∗ **ucc_p2p_conn_t**

8.7.3.7 typedef void∗ **ucc_context_addr_h**

8.7.3.8 typedef size_t **ucc_context_addr_len_t**

8.7.4 Enumeration Type Documentation

8.7.4.1 enum **ucc_team_params_field**

Enumerator

UCC_TEAM_PARAM_FIELD_ORDERING
UCC_TEAM_PARAM_FIELD_OUTSTANDING_COLLS
UCC_TEAM_PARAM_FIELD_EP
UCC_TEAM_PARAM_FIELD_EP_LIST
UCC_TEAM_PARAM_FIELD_EP_RANGE
UCC_TEAM_PARAM_FIELD_TEAM_SIZE
UCC_TEAM_PARAM_FIELD_SYNC_TYPE
UCC_TEAM_PARAM_FIELD_OOB
UCC_TEAM_PARAM_FIELD_P2P_CONN
UCC_TEAM_PARAM_FIELD_MEM_PARAMS
UCC_TEAM_PARAM_FIELD_EP_MAP
UCC_TEAM_PARAM_FIELD_ID
UCC_TEAM_PARAM_FIELD_FLAGS

8.7.4.2 enum **ucc_team_attr_field**

Enumerator

UCC_TEAM_ATTR_FIELD_POST_ORDERING
UCC_TEAM_ATTR_FIELD_OUTSTANDING_CALLS
UCC_TEAM_ATTR_FIELD_EP

UCC_TEAM_ATTR_FIELD_EP_RANGE

UCC_TEAM_ATTR_FIELD_SYNC_TYPE

UCC_TEAM_ATTR_FIELD_MEM_PARAMS

UCC_TEAM_ATTR_FIELD_SIZE

UCC_TEAM_ATTR_FIELD_EPS

8.7.4.3 enum **ucc_team_flags**

Enumerator

UCC_TEAM_FLAG_COLL_WORK_BUFFER

8.7.4.4 enum **ucc_post_ordering_t**

Enumerator

UCC_COLLECTIVE_POST_ORDERED When set to this value, the collective participants shall post the operation in the same order.

UCC_COLLECTIVE_POST_UNORDERED When set to this value, the collective participants shall post the operation in any order.

UCC_COLLECTIVE_INIT_ORDERED When set to this value, the collective participants shall initialize the operation in the same order.

UCC_COLLECTIVE_INIT_UNORDERED When set to this value, the collective participants shall initialize the operation in any order.

UCC_COLLECTIVE_INIT_AND_POST_ORDERED When set to this value, the collective participants shall initialize and post the operation in the same order.

UCC_COLLECTIVE_INIT_AND_POST_UNORDERED When set to this value, the collective participants shall initialize and post the operation in any order.

8.7.4.5 enum **ucc_ep_range_type_t**

Enumerator

UCC_COLLECTIVE_EP_RANGE_CONTIG

UCC_COLLECTIVE_EP_RANGE_NONCONTIG

8.7.4.6 enum **ucc_ep_map_type_t**

Enumerator

UCC_EP_MAP_FULL The ep range of the team spans all eps from a context.

UCC_EP_MAP_STRIDED The ep range of the team can be described by the 2 values: start, stride.

UCC_EP_MAP_ARRAY The ep range is given as an array of intergers that map the ep in the team to the team_context rank.

UCC_EP_MAP_CB The ep range mapping is defined as callback provided by the UCC user.

8.8 Team abstraction routines

Functions

- 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)

  The routine is a method to create the team.
- ucc_status_t ucc_team_create_test (ucc_team_h team)

  The routine queries the status of the team creation operation.
- ucc_status_t ucc_team_destroy (ucc_team_h team)

  The team frees the team handle.
- ucc_status_t ucc_team_get_attr (ucc_team_h team, ucc_team_attr_t ∗team_attr)

  The routine returns the attributes of the team.
- ucc_status_t ucc_team_create_from_parent (uint64_t my_ep, uint32_t included, ucc_team_↩ h parent_team, ucc_team_h ∗new_team)

  The routine creates a new team from the parent team.

8.8.1 Detailed Description

Team create and management routines

8.8.2 Function Documentation

8.8.2.1 **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)

Parameters

in	*contexts*	Communication contexts abstracting the resources
in	*num_contexts*	Number of contexts passed for the create operation
in	*team_params*	User defined configurations for the team
out	*new_team*	Team handle

Description

ucc_team_create_post is a nonblocking collective operation to create the team handle. Overlapping of multiple ucc_team_create_post operations are invalid. The post 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 ucc_status_t is returned.

Returns

Error code as defined by ucc_status_t

8.8.2.2 **ucc_status_t** ucc_team_create_test (**ucc_team_h** team)

Parameters

in	*team*	Team handle to test

Description

ucc_team_create_test routines tests the status of team handle. If required it can progress the communication but cannot block on the communications. On error, the team handle becomes invalid, user is responsible to call ucc_team_destroy to destroy team and free allocated resources.

Returns

Error code as defined by ucc_status_t

8.8.2.3 **ucc_status_t** ucc_team_destroy (**ucc_team_h** team)

Parameters

in	*team*	Destroy previously created team and release all resources associated with it.

Description

ucc_team_destroy is a nonblocking collective operation to release all resources associated with the team handle, and destroy the team handle. It is invalid to post a collective operation after the ucc_team_destroy operation. It is invalid to call ucc_team_destroy operation while ucc_team_create_post is in progress. It is the user's responsibility to ensure there is one outstanding ucc_team_create_post or ucc_team_destroy operation is in progress.

Returns

Error code as defined by ucc_status_t

8.8.2.4 **ucc_status_t** ucc_team_get_attr (**ucc_team_h** team, **ucc_team_attr_t** ∗ team_attr)

Parameters

in	*team*	Team handle
out	*team_attr*	Attributes of the team

Description

ucc_team_get_attr routine queries the team handle attributes. The attributes of the team handle are described by the team attributes ucc_team_attr_t

Returns

Error code as defined by ucc_status_t

8.8.2.5 **ucc_status_t** ucc_team_create_from_parent (uint64_t my_ep, uint32_t included, **ucc_team_h** parent_team, **ucc_team_h** ∗ new_team)

Parameters

in	*my_ep*	Endpoint of the process/thread calling the split operation
in	*parent_team*	Parent team handle from which a new team handle is created
in	*included*	Variable indicating whether a process/thread participates in the newly created team; value 1 indicates the participation and value 0 indicates otherwise
out	*new_team*	Pointer to the new team handle

Description

ucc_team_create_from_parent is a nonblocking collective operation, which creates a new team from the parent team. If a participant intends to participate in the new team, it passes a TRUE value for the "included" parameter. Otherwise, it passes FALSE. The routine returns immediately after the post-operation. To learn the completion of the team create operation, the ucc_team_create_test operation is used.

Returns

Error code as defined by ucc_status_t

8.9 Collective operations data-structures

Data Structures

- struct ucc_coll_buffer_info_v
- struct ucc_coll_buffer_info
- struct ucc_coll_callback

  UCC collective completion callback.

Typedefs

- typedef enum ucc_memory_type ucc_memory_type_t
- typedef struct ucc_coll_buffer_info_v ucc_coll_buffer_info_v_t
- typedef struct ucc_coll_buffer_info ucc_coll_buffer_info_t
- typedef struct ucc_coll_req ∗ ucc_coll_req_h

  UCC collective request handle.
- typedef struct ucc_coll_callback ucc_coll_callback_t

  UCC collective completion callback.
- typedef uint64_t ucc_count_t

  Count datatype to support both small (32 bit) and large counts (64 bit)
- typedef uint64_t ucc_aint_t

  Datatype to support both small (32 bit) and large address offsets (64 bit)
- typedef uint16_t ucc_coll_id_t

  Datatype for collective tags.

Enumerations

- enum ucc_memory_type {
  UCC_MEMORY_TYPE_HOST,
  UCC_MEMORY_TYPE_CUDA,
  UCC_MEMORY_TYPE_CUDA_MANAGED,
  UCC_MEMORY_TYPE_ROCM,
  UCC_MEMORY_TYPE_ROCM_MANAGED,
  UCC_MEMORY_TYPE_LAST,
  UCC_MEMORY_TYPE_UNKNOWN = UCC_MEMORY_TYPE_LAST }
- enum ucc_coll_args_flags_t {
  UCC_COLL_ARGS_FLAG_IN_PLACE = UCC_BIT(0),
  UCC_COLL_ARGS_FLAG_PERSISTENT = UCC_BIT(1),
  UCC_COLL_ARGS_FLAG_COUNT_64BIT = UCC_BIT(2),
  UCC_COLL_ARGS_FLAG_DISPLACEMENTS_64BIT = UCC_BIT(3),
  UCC_COLL_ARGS_FLAG_CONTIG_SRC_BUFFER = UCC_BIT(4),
  UCC_COLL_ARGS_FLAG_CONTIG_DST_BUFFER = UCC_BIT(5),
  UCC_COLL_ARGS_FLAG_TIMEOUT = UCC_BIT(6),
  UCC_COLL_ARGS_FLAG_MEM_MAPPED_BUFFERS = UCC_BIT(7) }
- enum ucc_coll_args_hints_t {
  UCC_COLL_ARGS_HINT_OPTIMIZE_OVERLAP_CPU = UCC_BIT(24),
  UCC_COLL_ARGS_HINT_OPTIMIZE_OVERLAP_GPU = UCC_BIT(25),
  UCC_COLL_ARGS_HINT_OPTIMIZE_LATENCY = UCC_BIT(26),
  UCC_COLL_ARGS_HINT_CONTIG_SRC_BUFFER = UCC_COLL_ARGS_FLAG_CONTIG_S↩
  RC_BUFFER,
  UCC_COLL_ARGS_HINT_CONTIG_DST_BUFFER = UCC_COLL_ARGS_FLAG_CONTIG_↩
  DST_BUFFER }
- enum ucc_error_type_t {
  UCC_ERR_TYPE_LOCAL = 0,
  UCC_ERR_TYPE_GLOBAL = 1 }

- enum ucc_coll_args_field {
  UCC_COLL_ARGS_FIELD_FLAGS = UCC_BIT(0),
  UCC_COLL_ARGS_FIELD_TAG = UCC_BIT(1),
  UCC_COLL_ARGS_FIELD_CB = UCC_BIT(2),
  UCC_COLL_ARGS_FIELD_GLOBAL_WORK_BUFFER = UCC_BIT(3),
  UCC_COLL_ARGS_FIELD_ACTIVE_SET = UCC_BIT(4) }

8.9.1 Detailed Description

Data-structures associated with collective operation creation, progress, and finalize.

8.9.2 Data Structure Documentation

8.9.2.1 struct ucc_coll_buffer_info_v

Data Fields

void ∗	buffer	Starting address of the send/recv buffer
ucc_count_t ∗	counts	Array of counts of type ucc_count_t describing the total number of elements
ucc_aint_t ∗	displacements	Displacement array of team size and type ucc_aint_t. Entry i specifies the displacement relative to the start address for the incoming data(outgoing data) for the team member i. For send buffer the data is fetched from this displacement and for receive buffer the incoming data is placed at this displacement.
ucc_datatype_t	datatype	Datatype of each buffer element
ucc_memory_type_t	mem_type	Memory type of buffer as defined by ucc_memory_type

8.9.2.2 struct ucc_coll_buffer_info

Data Fields

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 ucc_memory_type

8.9.3 Typedef Documentation

8.9.3.1 typedef enum **ucc_memory_type ucc_memory_type_t**

8.9.3.2 typedef struct **ucc_coll_buffer_info_v ucc_coll_buffer_info_v_t**

8.9.3.3 typedef struct **ucc_coll_buffer_info ucc_coll_buffer_info_t**

8.9.3.4 typedef struct **ucc_coll_req∗ ucc_coll_req_h**

The UCC request handle is an opaque handle created by the library during the invocation of the collective operation. The request may be used to learn the status of the collective operation, progress, or complete the collective operation.

8.9.3.5 typedef struct **ucc_coll_callback ucc_coll_callback_t**

The callback is invoked whenever the collective operation is completed. It is not allowed to call UCC APIs from the completion callback except for ucc_collective_finalize.

8.9.3.6 typedef uint64_t **ucc_count_t**

8.9.3.7 typedef uint64_t **ucc_aint_t**

8.9.3.8 typedef uint16_t **ucc_coll_id_t**

8.9.4 Enumeration Type Documentation

8.9.4.1 enum **ucc_memory_type**

Enumerator

UCC_MEMORY_TYPE_HOST Default system memory

UCC_MEMORY_TYPE_CUDA NVIDIA CUDA memory

UCC_MEMORY_TYPE_CUDA_MANAGED NVIDIA CUDA managed memory

UCC_MEMORY_TYPE_ROCM AMD ROCM memory

UCC_MEMORY_TYPE_ROCM_MANAGED AMD ROCM managed system memory

UCC_MEMORY_TYPE_LAST

UCC_MEMORY_TYPE_UNKNOWN

8.9.4.2 enum **ucc_coll_args_flags_t**

Enumerator

UCC_COLL_ARGS_FLAG_IN_PLACE If set, the output buffer is identical to the input buffer.

UCC_COLL_ARGS_FLAG_PERSISTENT If set, the collective is considered persistent. Only, the persistent collective can be called multiple times with the same request.

UCC_COLL_ARGS_FLAG_COUNT_64BIT If set, the count is 64bit, otherwise, it is 32 bit.

UCC_COLL_ARGS_FLAG_DISPLACEMENTS_64BIT If set, the displacement is 64bit, otherwise, it is 32 bit.

UCC_COLL_ARGS_FLAG_CONTIG_SRC_BUFFER If set, the src buffer is considered contiguous. Particularly, useful for alltoallv operation.

UCC_COLL_ARGS_FLAG_CONTIG_DST_BUFFER If set, the dst buffer is considered contiguous. Particularly, useful for alltoallv operation.

UCC_COLL_ARGS_FLAG_TIMEOUT If set and the elapsed time after ucc_collective_post (or ucc_collective_triggered_post) is greater than ucc_coll_args_t::timeout, the library returns UC↩ C_ERR_TIMED_OUT on the calling thread. Note, the status is not guaranteed to be global on all the processes participating in the collective.

UCC_COLL_ARGS_FLAG_MEM_MAPPED_BUFFERS If set, both src and dst buffers reside in a memory mapped region. Useful for one-sided collectives.

8.9.4.3 enum **ucc_coll_args_hints_t**

Enumerator

UCC_COLL_ARGS_HINT_OPTIMIZE_OVERLAP_CPU When the flag is set, the user prefers the library to choose an algorithm implementation optimized for the best overlap of CPU resources.

UCC_COLL_ARGS_HINT_OPTIMIZE_OVERLAP_GPU When the flag is set, the user prefers the library to choose an algorithm implementation optimized for the best overlap of GPU resources.

UCC_COLL_ARGS_HINT_OPTIMIZE_LATENCY When the flag is set, the user prefers the library to choose an algorithm implementation optimized for the latency.

UCC_COLL_ARGS_HINT_CONTIG_SRC_BUFFER When the flag is set, the source buffer is contiguous.

UCC_COLL_ARGS_HINT_CONTIG_DST_BUFFER When the flag is set, the destination buffer is contiguous.

8.9.4.4 enum **ucc_error_type_t**

Enumerator

UCC_ERR_TYPE_LOCAL

UCC_ERR_TYPE_GLOBAL

8.9.4.5 enum **ucc_coll_args_field**

Enumerator

UCC_COLL_ARGS_FIELD_FLAGS

UCC_COLL_ARGS_FIELD_TAG

UCC_COLL_ARGS_FIELD_CB

UCC_COLL_ARGS_FIELD_GLOBAL_WORK_BUFFER

UCC_COLL_ARGS_FIELD_ACTIVE_SET

8.10 Collective Operations

Data Structures

- struct ucc_coll_args

  Structure representing arguments for the collective operations. More...
- union ucc_coll_args.src
- union ucc_coll_args.dst
- struct ucc_coll_args.active_set

Typedefs

- typedef struct ucc_coll_args ucc_coll_args_t

  Structure representing arguments for the collective operations.
- typedef struct ucc_mem_handle * ucc_mem_h

  UCC memory handle.

Functions

- ucc_status_t ucc_collective_init (ucc_coll_args_t *coll_args, ucc_coll_req_h *request, ucc_team_h team)

  The routine to initialize a collective operation.
- ucc_status_t ucc_collective_post (ucc_coll_req_h request)

  The routine to post a collective operation.
- ucc_status_t ucc_collective_init_and_post (ucc_coll_args_t *coll_args, ucc_coll_req_h *request, ucc_team_h team)

  The routine to initialize and post a collective operation.
- static ucc_status_t ucc_collective_test (ucc_coll_req_h request)

  The routine to query the status of the collective operation.
- ucc_status_t ucc_collective_finalize (ucc_coll_req_h request)

  The routine to release the collective operation associated with the request object.

8.10.1 Detailed Description

Collective operations invocation and progress

8.10.2 Data Structure Documentation

8.10.2.1 struct ucc_coll_args

Description

ucc_coll_args_t defines the parameters that can be used to customize the collective operation. The "mask" bit array fields are defined by ucc_coll_args_field. The bits in "mask" bit array is defined by ucc_coll_args_field, which correspond to fields in structure ucc_coll_args_t. The valid fields of the structure are specified by setting the corresponding bit to "1" in the bit-array "mask".

The collective operation is selected by field "coll_type" which must be always set by user. If allreduce or * reduce operation is selected, the type of reduction is selected by the field * "predefined_reduction_op" or "custom_reduction_op". For unordered collective operations, the user-provided "tag" value orders the collective operation. For rooted collective operations such as reduce, scatter, gather, fan-in, and fan-out, the "root" field must be provided by user and specify the participant endpoint value. The user can request either "local" or "global" error information using the "error_type" field.

Information about user buffers used for collective operation must be specified according to the "coll_type".

Data Fields

uint64_t	mask	
ucc_coll_type_t	coll_type	Type of collective operation
union ucc_coll_args	src	
union ucc_coll_args	dst	
ucc_reduction_op_t	op	Predefined reduction operation, if reduce, allreduce, reduce_scatter operation is selected. The field is only specified for collectives that use pre-defined datatypes
uint64_t	flags	Provide flags and hints for the collective operations
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
void *	global_work_buffer	User allocated scratchpad buffer for one-sided collectives. The buffer provided should be at least the size returned by ucc_context_get_attr with the field mask - UCC_CONTEXT_ATTR_FIELD_WORK_BUFFER_SIZE set to 1. The buffer must be initialized to 0.
ucc_coll_callback_t	cb	
double	timeout	Timeout in seconds
struct ucc_coll_args	active_set	

8.10.2.2 union ucc_coll_args.src

Data Fields

ucc_coll_buffer_info_t	info	Buffer info for the collective
ucc_coll_buffer_info_v_t	info_v	Buffer info for the collective

8.10.2.3 union ucc_coll_args.dst

Data Fields

ucc_coll_buffer_info_t	info	Buffer info for the collective

ucc_coll_↩ buffer_info_v↩ _t	info_v	Buffer info for the collective

8.10.2.4 struct ucc_coll_args.active_set

Data Fields

uint64_t	start	
int64_t	stride	
uint64_t	size	

8.10.3 Typedef Documentation

8.10.3.1 typedef struct **ucc_coll_args ucc_coll_args_t**

Description

ucc_coll_args_t defines the parameters that can be used to customize the collective operation. The "mask" bit array fields are defined by ucc_coll_args_field. The bits in "mask" bit array is defined by ucc↩_coll_args_field, which correspond to fields in structure ucc_coll_args_t. The valid fields of the structure are specified by setting the corresponding bit to "1" in the bit-array "mask".

The collective operation is selected by field "coll_type" which must be always set by user. If allreduce or ∗ reduce operation is selected, the type of reduction is selected by the field ∗ "predefined_reduction_op" or "custom_reduction_op". For unordered collective operations, the user-provided "tag" value orders the collective operation. For rooted collective operations such as reduce, scatter, gather, fan-in, and fan-out, the "root" field must be provided by user and specify the participant endpoint value. The user can request either "local" or "global" error information using the "error_type" field.

Information about user buffers used for collective operation must be specified according to the "coll_↩type".

8.10.3.2 typedef struct ucc_mem_handle∗ **ucc_mem_h**

The UCC memory handle is an opaque handle created by the library representing the buffer and address.

8.10.4 Function Documentation

8.10.4.1 **ucc_status_t** ucc_collective_init (**ucc_coll_args_t** ∗ coll_args, **ucc_coll_req_h** ∗ request, **ucc_team_h** team)

Parameters

in	*coll_args*	Collective arguments descriptor
out	*request*	Request handle representing the collective operation
in	*team*	Team handle

Description

ucc_collective_init is a collective initialization operation, where all participants participate. The user provides all information required to start and complete the collective operation, which includes the input and output buffers, operation type, team handle, size, and any other hints for optimization. On success, the request handle is created and returned. On error, the request handle is not created and the appropriate error code is returned. On return, the ownership of buffers is transferred to the user. If modified, the results of collective operations posted on the request handle are undefined.

Returns

Error code as defined by ucc_status_t

8.10.4.2 **ucc_status_t** ucc_collective_post (**ucc_coll_req_h** request)

Parameters

in	*request*	Request handle

Description

ucc_collective_post routine posts the collective operation. It does not require synchronization between the participants for the post operation. On error, request handle becomes invalid, user is responsible to call ucc_collective_finalize to free allocated resources.

Returns

Error code as defined by ucc_status_t

8.10.4.3 **ucc_status_t** ucc_collective_init_and_post (**ucc_coll_args_t** ∗ coll_args, **ucc_coll_req_h** ∗ request, **ucc_team_h** team)

Parameters

out	*request*	Request handle representing the collective operation
in	*coll_args*	Collective arguments descriptor
in	*team*	Input Team

Description

ucc_collective_init_and_post initializes the collective operation and also posts the operation.

Note

: The ucc_collective_init_and_post can be implemented as a combination of ucc_collective_init and ucc_collective_post routines.

Returns

Error code as defined by ucc_status_t

8.10.4.4 static **ucc_status_t** ucc_collective_test (**ucc_coll_req_h** request) [inline], [static]

Parameters

in	*request*	Request handle

Description

ucc_collective_test tests and returns the status of collective operation. On error, request handle becomes invalid, user is responsible to call ucc_collective_finalize to free allocated resources.

Returns

Error code as defined by ucc_status_t

8.10.4.5 **ucc_status_t** ucc_collective_finalize (**ucc_coll_req_h** request)

Parameters

in	*request*	- Request handle

Description

ucc_collective_finalize operation releases all resources associated with the collective operation represented by the request handle. In UCC_THREAD_MULTIPLE mode, the user is responsible for ensuring that ucc_↩ collective_finalize is called after the status is UCC_OK and after completing the execution of any callback registered with ucc_coll_args_t.

Returns

Error code as defined by ucc_status_t

8.11 Events and Triggered operations' data-structures

Data Structures

- struct ucc_event
- struct ucc_ee_params

Typedefs

- typedef enum ucc_event_type ucc_event_type_t
- typedef enum ucc_ee_type ucc_ee_type_t
- typedef struct ucc_event ucc_ev_t
- typedef struct ucc_ee_params ucc_ee_params_t
- typedef struct ucc_ee ∗ ucc_ee_h

  UCC execution engine handle.

Enumerations

- enum ucc_event_type {
  UCC_EVENT_COLLECTIVE_POST = UCC_BIT(0),
  UCC_EVENT_COLLECTIVE_COMPLETE = UCC_BIT(1),
  UCC_EVENT_COMPUTE_COMPLETE = UCC_BIT(2),
  UCC_EVENT_OVERFLOW = UCC_BIT(3) }
- enum ucc_ee_type {
  UCC_EE_FIRST = 0,
  UCC_EE_CUDA_STREAM = UCC_EE_FIRST,
  UCC_EE_CPU_THREAD,
  UCC_EE_ROCM_STREAM,
  UCC_EE_LAST,
  UCC_EE_UNKNOWN = UCC_EE_LAST }

8.11.1 Detailed Description

Data-structures associated with event-driven collective execution

8.11.2 Data Structure Documentation

8.11.2.1 struct ucc_event

Data Fields

ucc_event_type_t	ev_type	
void ∗	ev_context	
size_t	ev_context_size	
ucc_coll_req_h	req	

8.11.2.2 struct ucc_ee_params

Data Fields

ucc_ee_type_t	ee_type	
void *	ee_context	
size_t	ee_context_size	

8.11.3 Typedef Documentation

8.11.3.1 typedef enum **ucc_event_type ucc_event_type_t**

8.11.3.2 typedef enum **ucc_ee_type ucc_ee_type_t**

8.11.3.3 typedef struct **ucc_event ucc_ev_t**

8.11.3.4 typedef struct **ucc_ee_params ucc_ee_params_t**

8.11.3.5 typedef struct ucc_ee* **ucc_ee_h**

The UCC execution engine handle is an opaque handle created by the library representing the execution context and related queues.

8.11.4 Enumeration Type Documentation

8.11.4.1 enum **ucc_event_type**

Enumerator

UCC_EVENT_COLLECTIVE_POST
UCC_EVENT_COLLECTIVE_COMPLETE
UCC_EVENT_COMPUTE_COMPLETE
UCC_EVENT_OVERFLOW

8.11.4.2 enum **ucc_ee_type**

Enumerator

UCC_EE_FIRST
UCC_EE_CUDA_STREAM
UCC_EE_CPU_THREAD
UCC_EE_ROCM_STREAM
UCC_EE_LAST
UCC_EE_UNKNOWN

8.12 Events and Triggered Operations

Functions

- ucc_status_t ucc_ee_create (ucc_team_h team, const ucc_ee_params_t *params, ucc_ee_h *ee)

  The routine creates the execution context for collective operations.
- ucc_status_t ucc_ee_destroy (ucc_ee_h ee)

  The routine destroys the execution context created for collective operations.
- ucc_status_t ucc_ee_get_event (ucc_ee_h ee, ucc_ev_t **ev)

  The routine gets the event from the event queue.
- ucc_status_t ucc_ee_ack_event (ucc_ee_h ee, ucc_ev_t *ev)

  The routine acks the events from the event queue.
- ucc_status_t ucc_ee_set_event (ucc_ee_h ee, ucc_ev_t *ev)

  The routine to set the event to the tail of the queue.
- ucc_status_t ucc_ee_wait (ucc_ee_h ee, ucc_ev_t *ev)

  The routine blocks the calling thread until there is an event on the queue.
- ucc_status_t ucc_collective_triggered_post (ucc_ee_h ee, ucc_ev_t *ee_event)

  The routine posts the collective operation on the execution engine, which is launched on the event.

8.12.1 Detailed Description

Event-driven Collective Execution

8.12.2 Function Documentation

8.12.2.1 **ucc_status_t** ucc_ee_create (**ucc_team_h** team, const **ucc_ee_params_t** * params, **ucc_ee_h** * ee)

Parameters

in	*team*	Team handle
in	*params*	User provided params to customize the execution engine
out	*ee*	Execution engine handle

Description

ucc_ee_create creates the execution engine. It enables event-driven collective execution. ucc_ee_params_t allows the execution engine to be configured to abstract either GPU and CPU threads. The execution engine is created and coupled with the team. There can be many execution engines coupled to the team. However, attaching the same execution engine to multiple teams is not allowed. The execution engine is created after the team is created and destroyed before the team is destroyed. It is the user's responsibility to destroy the execution engines before the team. If the team is destroyed before the execution engine is destroyed, the result is undefined.

Returns

Error code as defined by ucc_status_t

8.12.2.2 **ucc_status_t** ucc_ee_destroy (**ucc_ee_h** ee)

Parameters

in	*ee*	Execution engine handle

Description

ucc_ee_destroy releases the resources attached with the execution engine and destroys the execution engine. All events and triggered operations related to this ee are invalid after the destroy operation. To avoid race between the creation and destroying the execution engine, for a given ee, the ucc_ee_create and ucc_ee_destroy must be invoked from the same thread.

Returns

Error code as defined by ucc_status_t

8.12.2.3 **ucc_status_t** ucc_ee_get_event (**ucc_ee_h** ee, **ucc_ev_t** ** ev)

Parameters

in	*ee*	Execution engine handle
out	*ev*	Event structure fetched from the event queue

Description

ucc_ee_get_event fetches the events from the execution engine. If there are no events posted on the ee, it returns immediately without waiting for events. All events must be acknowledged using the ucc_ee_ack_event interface. The event acknowledged is destroyed by the library. An event fetched with ucc_ee_get_event but not acknowledged might consume resources in the library.

Returns

Error code as defined by ucc_status_t

8.12.2.4 **ucc_status_t** ucc_ee_ack_event (**ucc_ee_h** ee, **ucc_ev_t** * ev)

Parameters

in	*ee*	Execution engine handle
in	*ev*	Event to be acked

Description

An event acknowledged by the user using ucc_ee_ack_event is destroyed by the library. Any triggered operations on the event should be completed before calling this interface. The behavior is undefined if the user acknowledges the event while waiting on the event or triggering operations on the event.

Returns

Error code as defined by ucc_status_t

8.12.2.5 **ucc_status_t** ucc_ee_set_event (**ucc_ee_h** ee, **ucc_ev_t** * ev)

Parameters

in	*ee*	Execution engine handle
in	*ev*	Event structure fetched from the event queue

Description

ucc_ee_set_event sets the event on the execution engine. If the operations are waiting on the event when the user sets the event, the operations are launched. The events created by the user need to be destroyed by the user.

Returns

Error code as defined by ucc_status_t

8.12.2.6 **ucc_status_t** ucc_ee_wait (**ucc_ee_h** ee, **ucc_ev_t** * ev)

Parameters

in	*ee*	Execution engine handle
out	*ev*	Event structure fetched from the event queue

Description

The user thread invoking the ucc_ee_wait interface is blocked until an event is posted to the execution engine.

Returns

Error code as defined by ucc_status_t

8.12.2.7 **ucc_status_t** ucc_collective_triggered_post (**ucc_ee_h** ee, **ucc_ev_t** * ee_event)

Parameters

in	*ee*	Execution engine handle
in	*ee_event*	Event triggering the post operation

Description

ucc_collective_triggered_post allow the users to schedule a collective operation that executes in the future when an event occurs on the execution engine. On error, request handle associated with event becomes invalid, user is responsible to call ucc_collective_finalize to free allocated resources.

Returns

Error code as defined by ucc_status_t

8.13 Utility Operations

Enumerations

- enum ucc_config_print_flags_t {
  UCC_CONFIG_PRINT_CONFIG = UCC_BIT(0),
  UCC_CONFIG_PRINT_HEADER = UCC_BIT(1),
  UCC_CONFIG_PRINT_DOC = UCC_BIT(2),
  UCC_CONFIG_PRINT_HIDDEN = UCC_BIT(3) }

  Print configurations.
- enum ucc_status_t {
  UCC_OK = 0,
  UCC_INPROGRESS = 1,
  UCC_OPERATION_INITIALIZED = 2,
  UCC_ERR_NOT_SUPPORTED = -1,
  UCC_ERR_NOT_IMPLEMENTED = -2,
  UCC_ERR_INVALID_PARAM = -3,
  UCC_ERR_NO_MEMORY = -4,
  UCC_ERR_NO_RESOURCE = -5,
  UCC_ERR_NO_MESSAGE = -6,
  UCC_ERR_NOT_FOUND = -7,
  UCC_ERR_TIMED_OUT = -8,
  UCC_ERR_LAST = -100 }

  Status codes for the UCC operations.

Functions

- const char ∗ ucc_status_string (ucc_status_t status)

  Routine to convert status code to string.

8.13.1 Detailed Description

Helper functions to be used across the library

8.13.2 Enumeration Type Documentation

8.13.2.1 enum **ucc_config_print_flags_t**

Enumerator

UCC_CONFIG_PRINT_CONFIG
UCC_CONFIG_PRINT_HEADER
UCC_CONFIG_PRINT_DOC
UCC_CONFIG_PRINT_HIDDEN

8.13.2.2 enum **ucc_status_t**

Enumerator

UCC_OK
UCC_INPROGRESS Operation is posted and is in progress
UCC_OPERATION_INITIALIZED Operation initialized but not posted
UCC_ERR_NOT_SUPPORTED

UCC_ERR_NOT_IMPLEMENTED

UCC_ERR_INVALID_PARAM

UCC_ERR_NO_MEMORY

UCC_ERR_NO_RESOURCE

UCC_ERR_NO_MESSAGE General purpose return code without specific error

UCC_ERR_NOT_FOUND

UCC_ERR_TIMED_OUT

UCC_ERR_LAST

8.13.3 Function Documentation

8.13.3.1 const char∗ ucc_status_string (**ucc_status_t** status)

Chapter 9

Data Structure Documentation

9.1 ucc_coll_callback Struct Reference

UCC collective completion callback.

Data Fields

- void(∗ cb)(void ∗data, ucc_status_t status)
- void ∗ data

9.1.1 Detailed Description

The callback is invoked whenever the collective operation is completed. It is not allowed to call UCC APIs from the completion callback except for ucc_collective_finalize.

9.1.2 Field Documentation

9.1.2.1 void(∗ ucc_coll_callback::cb) (void ∗**data**, **ucc_status_t** status)

9.1.2.2 void∗ ucc_coll_callback::data

The documentation for this struct was generated from the following file:

- ucc_def.h

9.2 ucc_ep_map_cb Struct Reference

Data Fields

- uint64_t(∗ cb)(uint64_t ep, void ∗cb_ctx)
- void ∗ cb_ctx

9.2.1 Field Documentation

9.2.1.1 uint64_t(∗ ucc_ep_map_cb::cb) (uint64_t ep, void ∗**cb_ctx**)

9.2.1.2 void∗ ucc_ep_map_cb::cb_ctx

The documentation for this struct was generated from the following file:

- ucc.h

9.3 ucc_generic_dt_ops Struct Reference

UCC generic data type descriptor.

Data Fields

- uint64_t mask
- uint64_t flags
- size_t contig_size
- void ∗(∗ start_pack)(void ∗context, const void ∗buffer, size_t count)

  Start a packing request.
- void ∗(∗ start_unpack)(void ∗context, void ∗buffer, size_t count)

  Start an unpacking request.
- size_t(∗ packed_size)(void ∗state)

  Get the total size of packed data.
- size_t(∗ pack)(void ∗state, size_t offset, void ∗dest, size_t max_length)

  Pack data.
- ucc_status_t(∗ unpack)(void ∗state, size_t offset, const void ∗src, size_t length)

  Unpack data.
- void(∗ finish)(void ∗state)

  Finish packing/unpacking.
- struct {
  ucc_status_t(∗ cb)(const ucc_reduce_cb_params_t ∗params)
  void ∗ cb_ctx
  } reduce

  User-defined reduction callback.

9.3.1 Detailed Description

This structure provides a generic datatype descriptor that is used to create user-defined datatypes.

9.3.2 Field Documentation

9.3.2.1 uint64_t ucc_generic_dt_ops::mask

9.3.2.2 uint64_t ucc_generic_dt_ops::flags

9.3.2.3 size_t ucc_generic_dt_ops::contig_size

size of the datatype if UCC_GENERIC_DT_OPS_FLAG_CONTIG is set

The documentation for this struct was generated from the following file:

- ucc.h

9.4 ucc_generic_dt_ops.reduce Struct Reference

User-defined reduction callback.

Data Fields

- ucc_status_t(∗ cb)(const ucc_reduce_cb_params_t ∗params)
- void ∗ cb_ctx

9.4.1 Detailed Description

The pointer refers to user-defined reduction routine.

Parameters

in	*params*	reduction descriptor

9.4.2 Field Documentation

9.4.2.1

9.4.2.2

The documentation for this struct was generated from the following files:

9.5 ucc_oob_coll Struct Reference

OOB collective operation for creating the context.

Data Fields

- ucc_status_t(∗ allgather)(void ∗src_buf, void ∗recv_buf, size_t size, void ∗allgather_info, void ∗∗request)
- ucc_status_t(∗ req_test)(void ∗request)
- ucc_status_t(∗ req_free)(void ∗request)
- void ∗ coll_info
- uint32_t n_oob_eps
- uint32_t oob_ep

9.5.1 Field Documentation

9.5.1.1 **ucc_status_t**(∗ ucc_oob_coll::allgather) (void ∗src_buf, void ∗recv_buf, size_t size, void ∗allgather_info, void ∗∗request)

9.5.1.2 **ucc_status_t**(∗ ucc_oob_coll::req_test) (void ∗request)

9.5.1.3 **ucc_status_t**(∗ ucc_oob_coll::req_free) (void ∗request)

9.5.1.4 void∗ ucc_oob_coll::coll_info

9.5.1.5 uint32_t ucc_oob_coll::n_oob_eps

Number of endpoints participating in the oob operation (e.g., number of processes representing a ucc team)

9.5.1.6 uint32_t ucc_oob_coll::oob_ep

Integer value that represents the position of the calling processes in the given oob op: the data specified by "src_buf" will be placed at the offset "oob_ep∗size" in the "recv_buf". oob_ep must be uniq at every calling process and should be in the range [0:n_oob_eps).

The documentation for this struct was generated from the following file:

- ucc.h

9.6 ucc_team_p2p_conn Struct Reference

Data Fields

- int(∗ conn_info_lookup)(void ∗conn_ctx, uint64_t ep, ucc_p2p_conn_t ∗∗conn_info, void ∗request)
- int(∗ conn_info_release)(ucc_p2p_conn_t ∗conn_info)
- void ∗ conn_ctx
- ucc_status_t(∗ req_test)(void ∗request)
- ucc_status_t(∗ req_free)(void ∗request)

9.6.1 Field Documentation

9.6.1.1 int(∗ ucc_team_p2p_conn::conn_info_lookup) (void ∗**conn_ctx**, uint64_t ep, **ucc_p2p_conn_t** ∗∗conn_info, void ∗request)

9.6.1.2 int(∗ ucc_team_p2p_conn::conn_info_release) (**ucc_p2p_conn_t** ∗conn_info)

9.6.1.3 void∗ ucc_team_p2p_conn::conn_ctx

9.6.1.4 **ucc_status_t**(∗ ucc_team_p2p_conn::req_test) (void ∗request)

9.6.1.5 **ucc_status_t**(∗ ucc_team_p2p_conn::req_free) (void ∗request)

The documentation for this struct was generated from the following file:

- ucc.h

Index