Lawrence Berkeley National Laboratory

Title

UPC++ v1.0 Specification, Revision 2023.9.0

Permalink

https://escholarship.org/uc/item/4d9982jd

Authors

Bonachea, Dan
Kamil, Amir

Publication Date

2023-12-15

DOI

10.25344/S4J592

Peer reviewed

UPC++ v1.0 Specification
Revision 2023.9.0

UPC++ Specification Working Group
https://upcxx.lbl.gov

December 15, 2023

Abstract

UPC++ is a C++ library providing classes and functions that support Partitioned Global Address Space (PGAS) programming. The key communication facilities in UPC++ are one-sided Remote Memory Access (RMA) and Remote Procedure Call (RPC). All communication operations are syntactically explicit and default to non-blocking; asynchrony is managed through the use of futures, promises and continuation callbacks, enabling the programmer to construct a graph of operations to execute asynchronously as high-latency dependencies are satisfied. A global pointer abstraction provides system-wide addressability of shared memory, including host and accelerator memories. The parallelism model is primarily process-based, but the interface is thread-safe and designed to allow efficient and expressive use in multi-threaded applications. The interface is designed for extreme scalability throughout, and deliberately avoids design features that could inhibit scalability.

Citation

Authorship Acknowledgments

The UPC++ library is the result of collaboration between many individuals. The UPC++ library specification is developed by the UPC++ project at Lawrence Berkeley National Laboratory.

The current authors and maintainers of the UPC++ specification are:

> Dan Bonachea, Amir Kamil

Other current members of the UPC++ project team include:

> Johnny Corbino, Paul H. Hargrove, Colin A. MacLean,
> Damian Rouson, Daniel Waters

The following people have made significant contributions to prior revisions of this specification:

> John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin,
> Brian van Straalen

The UPC++ project has benefited from ideas and feedback from numerous members of the community. This notably includes:

> Hadia Ahmed, Rob Egan, Basilio B. Fraguela, Khaled Ibrahim,
> Bryce Adelstein Lelbach, Hongzhang Shan, Samuel Williams

The predecessor library, UPC++ v0.1 [9], was designed and developed by:

> Amir Kamil, Katherine Yelick, Yili Zheng

Recent Changes

The UPC++ library continues to evolve and improve in response to stakeholder requirements and feedback.

Notable changes in this specification relative to the 2023.3.0 revision are as follows:

1. Specify new `GpuDevice::uuid()` queries that return a device-specific hardware identifier associated with a GPU device.

2. Member function `future::ready()` has been renamed to `future::is_ready()`. The old function name is now deprecated and may be removed in a future revision.

3. Change the default argument value for `discharge()` and `progress_required()` from `top_persona_scope()` to `default_persona_scope()`. As such these functions now default to selecting all personas active with the calling thread.

4. Specify that attempts at internal-level progress from within the restricted context may be ignored.

5. Specify that invocation of `discharge()` is prohibited from within the restricted context.

6. Clarify that user-level progress executes available actions in an unspecified order.

7. Clarify details regarding how function objects provided to `future::then()` are invoked by the library.

8. Clarify details regarding how function objects provided to LPC are enlisted for execution, invoked by the library, and later destroyed.

9. Clarify constraints on any `future` returned by a function object that was passed to `persona::lpc()`.

10. Clarify the typing and semantics of the `future` returned by `persona::lpc()`.

11. Clarify details regarding how deserialized function objects provided to RPC are invoked by the library, and the ref-qualifiers permitted on `operator()` member functions thus invoked by the library.

The UPC++ 2023.9.0 software release implements all the changes described above. For details on the implementation, please consult https://upcxx.lbl.gov

Providing Feedback

Readers are encouraged to provide feedback and comments on this specification via one of the following channels:

1. Enter a new issue in the UPC++ Specification issue tracker, located at:
   https://upcxx-bugs.lbl.gov

2. Send email to upcxx@googlegroups.com – this is a semi-public forum for maintainers of UPC++ and other interested parties.

Contents

Chapter 1

Overview and Scope

1.1 Preliminaries

[1] `UPC++` is a C++ library providing classes and functions that support Partitioned Global Address Space (PGAS) programming. The project began in 2012 with a prototype labeled V0.1, described in the IPDPS14 paper by *Zheng et al.* [9]. This document describes a production version, V1.0, with the addition of several features and a new asynchronous API. For a peer-reviewed overview of the new version, see the IPDPS19 paper [5].

[2] Under the PGAS model, a distributed memory parallel computer is viewed abstractly as a collection of *processing elements*, an individual computing resource, each with *local memory* (see Fig. 1.1). A processing element is called a *process* in `UPC++`. The execution model of `UPC++` is SPMD and the number of `UPC++` processes is fixed during program execution.

[3] As with conventional C++ threads programming, processes can access their respective local memory via a pointer. However, the PGAS abstraction supports a global address space, which is allocated in *shared segments* distributed over the processes. A *global pointer* enables the UPC++ programmer to reference objects in the shared segments between processes as shown in Fig. 1.2. As with threads programming, accesses to shared objects made via global pointers may be subject to race conditions, and appropriate synchronization must be employed.

[4] `UPC++` global pointers are fundamentally different from conventional C-style pointers. A global pointer refers to a location in a shared segment. It cannot be dereferenced using the `*` operator, and it cannot be constructed by the address-of operator. Rather, there are more syntactically explicit methods for accomplishing these tasks. On the other hand,

1
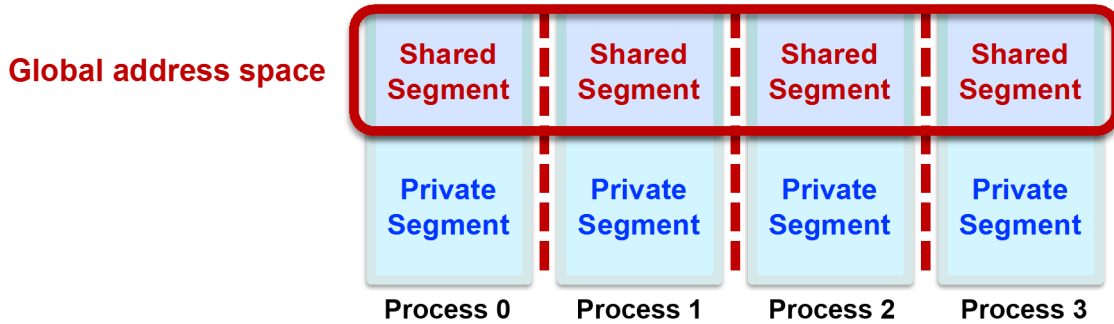
Figure 1.1: Abstract Machine Model of a PGAS program memory
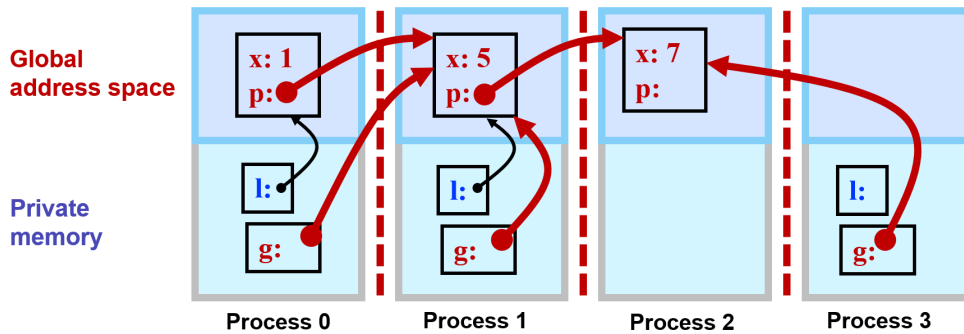
Figure 1.2: Global pointers and shared memory objects in a PGAS model

UPC++ global pointers *do* support some properties of a regular C pointer, such as pointer arithmetic and passing a pointer by value.

5 Notably, global pointers are used in *one-sided* communication: Remote Memory Access (RMA) operations similar to *memcpy* but across processes (Ch. 8), and in Remote Procedure Calls (RPC, Ch. 9). RPC enables the programmer to move computation to other processes, which is useful in managing irregular distributed data structures. These processes can push or pull data via global pointers. *Futures* and *Promises* (Ch. 5) are used to determine completion of communication or to schedule callbacks that respond to completion. Wherever possible, UPC++ engages low-level hardware support for communication and this capability is crucial to UPC++'s support of *lightweight communication*.

6 UPC++'s design philosophy is to encourage writing scalable, high-performance applications. UPC++ imposes certain restrictions in order to meet this goal. In particular, non-blocking communication is the default for nearly all operations defined in the API, and all communication is explicit. These two restrictions encourage the programmer to write code that is performant and make it more difficult to write code that is not. Conversely, UPC++ relaxes some restrictions found in models such as MPI; in particular, it does not impose an in-order

delivery requirement between separate communication operations. The added flexibility increases the possibility of overlapping communication and scheduling it appropriately.

7   UPC++ also avoids non-scalable constructs found in models such as UPC. For example, it does not support shared distributed arrays or shared scalars. Instead, it provides distributed objects, which can be used for similar purposes (Ch. 14). Distributed objects are useful in solving the *bootstrapping problem*, whereby processes need to distribute their local copies of global pointers to other processes. Though UPC++ does not directly provide multidimensional arrays, applications that use UPC++ may define them. To this end, UPC++ supports non-contiguous data transfers for regular, irregular and strided data (Ch. 15).

8   UPC++ does not create internal threads to manage progress. Therefore, UPC++ must manage all progress inside active calls to the library. The strengths of this approach include improved user-visibility into the resource requirements of UPC++ and better interoperability with software packages and their possibly restrictive threading requirements. The consequence, however, is that the user must be conscientious to balance the need for making progress against the application's need for CPU cycles. Chapter 10 discusses subtleties of managing progress and how an application can arrange for UPC++ to advance the state of asynchronous communication.

9   Processes may be grouped into teams (Ch. 11). A team can participate in collective operations. Teams are also the interface that UPC++ uses to advertise the shared memory capabilities of the underlying hardware and operating system. This enables a programmer to reason about hierarchical processor-memory organization, allowing an application to reduce its memory footprint. UPC++ supports remote atomic memory operations (Ch. 13). Atomics are useful in managing distributed queues, hash tables, and so on. However, UPC++ remote atomic operations are explicitly split-phased and handled somewhat differently from the process-scope atomics provided in C++11 std::atomic.

10 UPC++ supports memory kinds (Ch. 16), whereby the programmer can identify regions of memory requiring different access methods or having different performance properties, such as GPU device memory [8]. Global pointers can reference device memory, and UPC++ supports seamless data motion between any combination of host or device memory, whether local or remote.

1.2   Execution Model

1   The UPC++ state for each process contains internal unordered queues that are managed for the user. The UPC++ progress engine scans these queues for operations initiated by this process, as well as externally generated operations that target this process. The progress

engine is active inside `UPC++` calls only and is quiescent at other times, as there are no threads or background processes executing inside `UPC++`. This passive stance permits `UPC++` to be driven by any other execution model a user might choose. This universality does place a small burden on the user: calling into the `progress` function. `UPC++` relies on the user to make periodic calls into the `progress` function to ensure that `UPC++` operations are completed. `progress` is the mechanism by which the user loans `UPC++` a thread of execution to perform operations that target the given process. The user can determine that a specific operation completes by checking the status of its associated `future`, or by attaching a completion handler to the operation.

2 `UPC++` presents a *thread-aware* programming model. With a few exceptions, it generally assumes that only one thread of execution is interacting with any given library object at a time. The abstraction for thread-awareness in `UPC++` is the *persona*. A `future` produced by a thread of execution is associated with its persona, and transferring the `future` to another thread must be accompanied by transferring the underlying persona. Each process has a *master persona*, initially attached to the thread that calls `init`. Some `UPC++` operations, such as `barrier`, require a thread to have exclusive access to the master persona to call them. Thus, the programmer is responsible for ensuring synchronized access to both personas and memory.

1.3 Memory Model

1 The `UPC++` memory model differs from that of C++11 (and beyond) in that all updates are split-phased: every communication operation has a distinct initiate and wait step. Thus, RMA operations execute over a time interval, and the time intervals of successive operations that target the same datum must not overlap, or a data race will result.

2 `UPC++` differs from message passing in MPI in that it doesn't guarantee in-order delivery.[1] For example, if we overlap two successive RMA operations involving the same source and destination process, there are no guarantees regarding which will complete first. The same lack of implicit point-to-point ordering holds for all asynchronous operations (including RMA, RPC, remote atomics, etc). The only way to guarantee ordering is to apply explicit synchronization, e.g. issue a wait on a prior operation before initiating any dependent operation.

---

[1]MPI supports RMA, which is also unordered.

1.4   Common Requirements

1   Unless explicitly stated otherwise, the requirements in [res.on.arguments] in the C++ standard apply to `UPC++` functions as well. In particular, if a local or global pointer passed to a `UPC++` function is invalid for its intended use, the behavior of the function is undefined.

2   A number of functions accept a sequence of inputs delimited by two Iterator arguments, e.g. `[begin,end)`. Such functions have implicit preconditions that `end` must be reachable by incrementing `begin` zero or more times, and all accesses to the underlying objects in the sequence must be valid. Otherwise the behavior of the function is undefined.

3   `UPC++` functions may call into user code, including invoking constructors and destructors and running user-provided callbacks. If an exception propagates from user code into a `UPC++` function specified as `noexcept`, the behavior is undefined[2]. As indicated in §1.6, `UPC++` functions are implicitly declared `noexcept` unless specified otherwise. Thus, if user-provided code throws an exception that propagates outward into any UPC++ function whose specification does not include an *Exceptions* clause explicitly allowing this, the behavior is undefined. In particular, this means that unless otherwise specified, any specified constraint on a user-provided type (e.g., "must be MoveConstructible") also assumes the relevant operation does not actually throw an exception, otherwise behavior is undefined.

4   For `UPC++` functions with a *Precondition(s)* clause, violation of the preconditions results in undefined behavior.

1.5   Organization of this Document

1   This specification is intended to be a normative reference, not a tutorial on learning to use the library. A Programmer's Guide is available from https://upcxx.lbl.gov and is a good tutorial to gain a working understanding of the library.

2   The organization for the rest of the document is as follows. Chapter 2 discusses the process of starting up and closing down `UPC++`. Global pointers (Ch. 3) are fundamental to the PGAS model, and Chapter 4 discusses shared heap storage management. `UPC++` supports aggressively asynchronous communication and provides futures and promises (Ch. 5) to manage asynchronous operations and control flow. Chapter 6 discusses how C++ objects are serialized for communication. Chapter 7 describes the different completion models available for `UPC++` communication operations. Chapters 8 and 9 describe two core forms of

---

[2]This differs from standard C++, which specifies that `std::terminate` is called if an exception reaches the outermost block of a non-throwing function.

asynchronous one-sided communication, RMA and RPC, respectively. Chapter 10 discusses progress. Chapter 11 discusses teams, which are a means of organizing UPC++ processes, and Chapter 12 describes collective communication operations. Chapter 13 discusses atomic operations on shared memory. Chapter 14 discusses distributed objects. Chapter 15 discusses non-contiguous one-sided RMA transfers. Chapter 16 discusses memory kinds for device memory.

1.6 Conventions

1. All entities are declared by the header `upcxx/upcxx.hpp`, unless otherwise specified.

2. All library identifiers are in the `upcxx` namespace, unless otherwise qualified.

3. All functions are declared noexcept unless specifically called out.

4. The notation *cq* represents an optional `const` qualifier.

5. All instances of `size_t` and `ptrdiff_t` are illustrative shorthand for the corresponding fully-qualified `std::` type.

6. Except where otherwise specified, all library functions have a progress level (§10) guarantee of *progress level: none.*

1.7 Glossary

[1] **Affinity.** A binding of each location in a shared or device segment to a particular process (generally the process which allocated that shared object). Every byte of shared memory has affinity to exactly one process (at least logically).

[2] **C++ Concepts.** E.g. TriviallyCopyable. This document references C++ Concepts as defined in the C++14 standard [3] when specifying the semantics of types. However, compliant implementations are still possible within a compiler adhering to the earlier C++11 standard [2].

[3] **Collective.** A constraint placed on some language operations which requires evaluation of such operations to be matched across all participating processes. The behavior of collective operations is undefined unless all processes execute the same sequence of collective operations.

[4] A collective operation need not provide any actual synchronization between processes, unless otherwise noted. The collective requirement simply states a relative

ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any valid program. Some implementations may include unspecified synchronization between processes within collective operations, but programs must not rely upon the presence or absence of such unspecified synchronization for correctness.

5 **Collective object.** (16) A semantic binding of objects constructed and destroyed collectively by the processes in a team.

6 **Device.** (16) A physical device with storage that is distinct from main memory.

7 **Device segment.** (16) A region of storage associated with a device that is used to allocate objects that are accessible by any process.

8 **Enlist (Enlisted, Enlistment).** (10) A semantic action whereby a given continuation or callback is inserted into an unordered collection of actions destined for later execution during progress of a specified persona.

9 **Futures (and Promises).** (5) The primary mechanisms by which a `UPC++` application interacts with non-blocking operations. The semantics of futures and promises in `UPC++` differ from the those of standard C++. While futures in C++ facilitate communicating between threads, the intent of `UPC++` futures is solely to provide an interface for managing and composing non-blocking operations, and they cannot be used directly to communicate between threads or processes. A future is the interface through which the status of the operation can be queried and the results retrieved, and multiple future objects may be associated with the same promise. A future thus represents the consumer side of a non-blocking operation. A promise represents the producer side of the operation, and it is through the promise that the results of the operation are supplied and its dependencies fulfilled.

10 **Global pointer.** (3) The primary way to address memory in a shared memory segment of a `UPC++` program. Global pointers can themselves be stored in shared memory or otherwise passed between processes and retain their semantic meaning to any process.

11 **Local.** (11.2) Refers to an object or reference with affinity to a process in the local team.

12 **Operation completion.** (7) The condition where a communication operation is complete with respect to the initiating process, such that its effects are visible and that resources, such as source and destination memory regions, are no longer in use by `UPC++`.

13 **Persona.** (10.4) The abstraction for thread-awareness in `UPC++`. A `UPC++` persona object represents a collection of `UPC++`-internal state usually attributed to a single thread. By making it a proper construct, `UPC++` allows a single OS thread to switch between

multiple application-defined roles for processing notifications. Personas act as the receivers for notifications generated by the `UPC++` runtime.

[14] **Private object.** An object outside the shared space that can be accessed only by the process that owns it (e.g. an object on the program stack).

[15] **Process.** (1) An OS process with associated system resources that is a member of a `UPC++` parallel job execution. `UPC++` uses a SPMD execution model, and the number of processes is fixed during a given program execution. The placement of processes across physical processors or NUMA domains is implementation-defined.

[16] **Progress.** (10) The means by which the application allows the `UPC++` runtime to advance the state of outstanding operations initiated by this or other processes, to ensure they eventually complete.

[17] **Rank.** (11) An integer index that identifies a unique `UPC++` process within a `UPC++` `team`.

[18] **Referentially transparent.** A routine that is is a pure function, where inputs alone determine the value returned by the function. For the same inputs, repeated calls to a referentially transparent function will always return the same result.

[19] **Remote.** Refers to an object or reference whose affinity is not local to the current process.

[20] **Remote Procedure Call.** A communication operation that injects a function call invocation into the execution stream of another process. These injections are one-sided, meaning the target process need not explicitly expect the incoming operation or perform any specific action to receive it, aside from invoking `UPC++` progress.

[21] **Serializable.** (6) A C++ type that is either TriviallySerializable, or that implements the `UPC++` class serialization interface.

[22] **Source completion.** The condition where a communication operation initiated by the current process has advanced to a point where serialization of the local source memory regions for the operation has occurred, and the contents of those regions can be safely overwritten or reclaimed without affecting the behavior of the ongoing operation. Source completion does not generally imply operation completion, and other effects of the operation (e.g., updating destination memory regions, or delivery to a remote process) may still be in-progress.

[23] **Shared segment.** A region of storage associated with a particular process that is used to allocate shared objects that are accessible by any process.

[24] **Team.** (11) A `UPC++` object representing an ordered set of processes. Each process in a team has a unique 0-based rank index.

²⁵ **Thread (or OS thread).** An independent stream of executing instructions with private state. A process may contain many threads (created by the application), and each is associated with at least one persona.

²⁶ **TriviallySerializable.** (6) A C++ type that is valid to serialize by making a byte copy of an object.

Chapter 2

Init and Finalize

2.1 Overview

1 The `init` function must be called before any other `UPC++` function can be invoked. This can happen anywhere in the program, so long as it appears before any `UPC++` calls that require the library to be in an initialized state. The call is *collective*, meaning every process in the parallel job must enter this function if any are to participate in `UPC++` operations. While `init` can be called more than once by each process in a program, only the first invocation will initialize `UPC++`, and the rest will merely increment the internal count of how many times `init` has been called. For each `init` call, a matching `finalize` call must eventually be made. `init` and `finalize` are not re-entrant and must be called by only a single thread of execution in each process. The thread that calls `init` has the *master persona* attached to it (see section 10.5.1 for more details of threading behavior). After the number of calls to `finalize` matches the number of calls to `init`, no `UPC++` function that requires the library to be in an initialized state can be invoked until `UPC++` is reinitialized by a subsequent call to `init`.

2 All `UPC++` operations require the library to be in an initialized state unless otherwise specified, and violating this requirement results in undefined behavior. Member functions, constructors, and destructors are included in the set of operations that require `UPC++` to be initialized, unless explicitly stated otherwise.

```
1 #include <upcxx/upcxx.hpp>
2 #include <iostream>
3 int main(int argc, char *argv[])
4 {
5   upcxx::init();                          // initialize UPC++
6
7   std::cout << "Hello World"
8     << " ranks:" << upcxx::rank_n()       // how many processes?
9     << " my rank: " << upcxx::rank_me()   // which process am I?
10    << std::endl;
11
12  upcxx::finalize();                       // finalize UPC++
13  return 0;
14 }
```

Figure 2.1: *HelloWorld.cpp* program

2.2   Hello World

1   A `UPC++` installation should be able to compile and execute the simple *Hello World* program shown in Figure 2.1. The output of *Hello World*, however, is platform-dependent and may vary between different runs, since there is no synchronization to order the output between processes. Depending on the nature of the buffering protocol of `stdout`, output from different processes may even be interleaved.

2.3   API Reference

```
1 void init();
```

2   *Precondition:* Called collectively by all processes in the parallel job. The master persona (§10.5.1) must appear in the persona stack of the calling thread if `UPC++` is in an already-initialized state.

3   If there have been no previous calls to `init`, or if all previous calls to `init` have had matching calls to `finalize`, then this routine initializes the `UPC++` library. Initialization of the library also pushes the master persona onto the active persona stack of the calling thread (§10.5.1).

4   Otherwise, leaves the library state and active persona stack unchanged.

5   *This function may be called when* `UPC++` *is in the uninitialized state.*

```
6  bool initialized ();
```

7   Returns whether or not `UPC++` is in the initialized state. `UPC++` is initialized if
       there has been at least one previous call to `init` that has not had a matching
       call to `finalize`.

8   *This function may be called when* `UPC++` *is in the uninitialized state.*

```
9  void finalize ();
```

10   *Precondition:* Called collectively by all processes in the parallel job. The master
       persona (§10.5.1) must appear in the persona stack of the calling thread, and
       `UPC++` must be in an already-initialized state.

11   If this call matches the call to `init` that placed `UPC++` in an initialized state,
       then this call uninitializes the `UPC++` library. Otherwise, this function does not
       alter the library's state.

12   Before uninitializing the `UPC++` library, `finalize` shall execute a (blocking)
       `barrier()` over team `world()`. If this call uninitializes the `UPC++` library while
       there are any asynchronous operations still in-flight (after the barrier), behav-
       ior is undefined. An operation is defined as in-flight if it was initiated but still
       requires internal-level or user-level progress from any persona on any process
       in the job before it can complete. It is left to the application to define and
       implement their own specific approach to ensuring quiescence of in-flight oper-
       ations. A potential quiescence API is being considered for future versions and
       feedback is encouraged.

13   *UPC++ progress level:* `user`

```
14  #define UPCXX_SPEC_VERSION 20230900L
```

15   A macro definition to an integer literal identifying the version of this specifi-
       cation. Implementations complying to this specification shall define the value
       shown above. It is intended that future versions of this specification will replace
       the value of this macro with a greater value.

```
16  #define UPCXX_VERSION /* implementation -defined */
```

17   A macro definition to an integer literal identifying the version of the imple-
       mentation. Values are implementation-defined, but are recommended to be
       monotonically non-decreasing for subsequent revisions of the same implemen-
       tation.

```
18 char *getenv_console(const char *env_var);
```

19   The `getenv_console` function searches an environment list, provided by the host environment, for a string that matches the string pointed to by `env_var`.

20   The function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `getenv_console` function. If the specified name cannot be found, a null pointer is returned.

21   It is unspecified whether and how functions that modify the POSIX environment (`std::setenv`, `std::unsetenv`, `std::putenv`, etc.) affect the return values of this function.

22   *Advice to users:* On some platforms, environment variables provided by the spawning console might not be propagated to the POSIX environment of all UPC++ processes. This function's semantics are the same as `std::getenv()`, except that if `env_var` was set in the environment of the spawning console, that value is instead returned. UPC++ programs are recommended to use this function instead of `std::getenv()` to help ensure portability.

23   *Note:* As with most library functions, this function requires UPC++ to be in an already-initialized state.

Chapter 3

Global Pointers

3.1 Overview

1 The `UPC++` `global_ptr` is the primary way to address memory in a remote shared memory segment of a `UPC++` program. The next chapter discusses how memory in the shared segment is allocated to the user.

2 As mentioned in Chapter 1, a global pointer is a handle that may not be dereferenced. This restriction follows from the design decision to prohibit implicit communication. Logically, a global pointer has two parts: a raw C++ pointer and an associated *affinity*, which is a binding of each location in a shared or device (Ch. 16) segment to a particular process (generally the process which allocated that shared object). In cases where the use of a `global_ptr` executes in a process that has direct load/store access to the memory of the `global_ptr` (i.e. `is_local` is `true`), we may extract the raw pointer component, and benefit from the reduced cost of employing a local reference rather than a global one. To this end, `UPC++` provides the `local()` function, which returns a raw C++ pointer. Calling `local()` on a `global_ptr` that references an address in a remote shared segment or a device location to which the caller does not have load/store access results in undefined behavior.

3 Global pointers have the following guarantees:

1. A `global_ptr<T, Kind>` is only valid if it is the null global pointer, it references a valid object, or it represents one element past the end of a valid array or non-array object.

2. Two global pointers compare equal if and only if they reference the same object, one past the end of the same array or non-array object, or are both null.

3. Equality of global pointers corresponds to observational equality, meaning that two global pointers which compare equal will produce equivalent behavior when interchanged.

4 These facts become important given that `UPC++` allows two processes which are local to each other to map the same memory into their own virtual address spaces but possibly with different virtual addresses. They also ensure that a global pointer can be viewed from any process to mean the same thing without need for translation.

5 Global pointers are parameterized by the kind of memory they can refer to. A global pointer of type `global_ptr<T, Kind>` can only refer to memory on devices described by `Kind`, and the referenced memory may be located on a device attached to a local or remote process. The default global pointer, `global_ptr<T, memory_kind::host>`, can only refer to host memory on a local or remote process. A `global_ptr<T, memory_kind::any>` can refer to either host memory or memory on any device associated with a local or remote process.

6 Most `UPC++` communication operations only operate on host memory, working on the default `global_ptr<T>`. Functions that work with device memory are additionally parameterized by memory kind, working with general types such as `global_ptr<T, Kind>`.

7 The type `global_ptr<T, Kind>` is implicitly convertible to `global_ptr<const T, Kind>`, even in contexts that require template deduction. Thus, a `global_ptr<T, Kind>` may be passed to any `UPC++` operation that requires a `global_ptr<const T, Kind>`.

8 On the other hand, an object of type `global_ptr<const T, Kind>` can only be converted to a `global_ptr<T, Kind>` by a call to `const_pointer_cast<T>()`.

3.2 API Reference

1 ```
using intrank_t = /* see below */;
```

2      A signed integer type that represents a `UPC++` rank ID.

```
3  enum class memory_kind {
     any = /* unspecified */,
     host = /* unspecified */,
     cuda_device = /* unspecified */,
     hip_device = /* unspecified */,
     ze_device = /* unspecified */
   };
```

4   Constants used with a global pointer to specify the kind of memory (Ch. 16)
       that may be referenced by the global pointer.

```
5  template<typename T, memory_kind Kind = memory_kind::host>
   struct global_ptr : global_ptr<const T, Kind>;
   template<typename T, memory_kind Kind>
   struct global_ptr<const T, Kind>;
```

6   *C++ Concepts:* DefaultConstructible, TriviallyCopyable, TriviallyDestructible,
       EqualityComparable, LessThanComparable, hashable

7   UPC++ *Concepts:* TriviallySerializable

8   T must not be qualified with `volatile`: `std::is_volatile<T>::value` must
       be false.

9   T must not be a reference type: `std::is_reference<T>::value` must be false.

10   T may be an incomplete type, but some member functions are specified to
       require T to be a complete type at invocation.

```
11 template<typename T, memory_kind Kind>
   class global_ptr {
     using element_type = T;
     using pointer_type = T*;
     // ...
   };
```

12   Member type aliases for the template parameter T and the underlying raw
       pointer type.

```
13 template<typename T, memory_kind Kind>
   static const memory_kind global_ptr<T, Kind>::kind = Kind;
```

14   Constant that has the same value as the Kind template parameter.

15
```
template < typename T , memory_kind Kind >
global_ptr <T , Kind >:: global_ptr ( std :: nullptr_t = nullptr );
```

16   Constructs a global pointer corresponding to a null pointer.

17   *This function may be called when* UPC++ *is in the uninitialized state.*

18
```
template < typename T >
template < memory_kind Kind >
global_ptr <T , memory_kind :: any >:: global_ptr (
                                 global_ptr <T , Kind > other );
```

19   Constructs a global pointer with kind `memory_kind::any` from an existing global pointer.

20
```
template < typename T , memory_kind Kind >
global_ptr <T , Kind >::~ global_ptr ();
```

21   Trivial destructor. Does not delete or otherwise reclaim the raw pointer that this global pointer is referencing.

22   *This function may be called when* UPC++ *is in the uninitialized state.*

23
```
template < typename T >
global_ptr <T > to_global_ptr (T* ptr );
```

24   *Precondition:* `ptr` is a null pointer, or a valid pointer to host memory such that the expression `*ptr` on the calling process yields a (possibly uninitialized) object of type `T` that resides within the shared segment of a process in the local team (§11.2) of the caller

25   Constructs a global pointer corresponding to the given raw pointer.

26
```
template < typename T >
global_ptr <T > try_global_ptr (T* ptr );
```

27   *Precondition:* `ptr` is a null pointer, or a valid pointer to host memory such that the expression `*ptr` on the calling process yields a (possibly uninitialized) object of type `T`

28   If the object referenced by `*ptr` resides within the shared segment of a process in the local team (§11.2) of the caller, returns a global pointer referencing that object. Otherwise returns a null pointer.

29 ```
    template < typename T, memory_kind Kind >
    memory_kind global_ptr <T, Kind >:: dynamic_kind () const ;
    ```

30   If `!is_null()`, returns the actual memory kind associated with the memory
        referenced by this pointer.

31   If `is_null()`, the result is unspecified.

32 ```
    template < typename T, memory_kind Kind >
    bool global_ptr <T, Kind >:: is_local () const ;
    ```

33      Returns whether or not the calling process has load/store access to the memory
        referenced by this pointer. Returns true if this is a null pointer, regardless of the
        context in which this query is called. Otherwise, the result is unspecified if this
        pointer targets device memory (i.e. `dynamic_kind() != memory_kind::host`).

34  ```
    template < typename T, memory_kind Kind >
    bool global_ptr <T, Kind >:: is_null () const ;
    ```

35   Returns whether or not this global pointer corresponds to the null value, mean-
        ing that it references no memory. This query is purely a function of the global
        pointer instance, it is not affected by the context in which it is called.

36 ```
    template < typename T, memory_kind Kind >
    [explicit] bool global_ptr <T, Kind >:: operator bool () const ;
    ```

37      Explicit conversion operator that returns `!is_null()`.

38  ```
    template < typename T, memory_kind Kind >
    T* global_ptr <T, Kind >:: local () const ;
    ```

39   *Precondition:* `this->is_local()`

40   Converts this global pointer into a raw pointer.

41
```
template<typename T, memory_kind Kind>
intrank_t global_ptr<T, Kind>::where() const;
```

42   Returns the rank in team `world()` of the process with affinity to the `T` object pointed-to by this global pointer. The return value for `where()` on a null global pointer is an implementation-defined value.

43   For a non-null device pointer (`dynamic_kind() != memory_kind::host`), returns the rank in team `world()` of the process that allocated the memory referenced by this pointer. The result is undefined if this pointer references unallocated memory.

44   This query is purely a function of the global pointer instance, it is not affected by the context in which it is called.

45
```
template<typename T, memory_kind Kind>
global_ptr<T, Kind>
    global_ptr<T, Kind>::operator+(ptrdiff_t diff) const;
template<typename T, memory_kind Kind>
global_ptr<T, Kind>
    operator+(ptrdiff_t diff, global_ptr<T, Kind> ptr);
template<typename T, memory_kind Kind>
global_ptr<T, Kind>&
    global_ptr<T, Kind>::operator+=(ptrdiff_t diff);
```

46   *Precondition:* T must be a complete type. Either `diff == 0`, or the global pointer is pointing to the `i`th element of an array of `N` elements, where `i` may be equal to `N`, representing a one-past-the-end pointer. At least one of the indices `i+diff` or `i+diff-1` must be a valid element of the same array. A pointer to a non-array object is treated as a pointer to an array of size 1.

47   If `diff == 0`, returns a copy of the global pointer. Otherwise produces a pointer that references the element that is at `diff` positions greater than the current element, or a one-past-the-end pointer if the last element of the array is at `diff-1` positions greater than the current.

48   `operator+=` modifies the `global_ptr` in-place and returns a reference to this pointer after the operation.

49   These routines are purely functions of their arguments, they are not affected by the context in which they are called.

```
50  template<typename T, memory_kind Kind>
    global_ptr<T, Kind>
        global_ptr<T, Kind>::operator-(ptrdiff_t diff) const;
    template<typename T, memory_kind Kind>
    global_ptr<T, Kind>&
        global_ptr<T, Kind>::operator-=(ptrdiff_t diff);
```

51 *Precondition:* `T` must be a complete type. Either `diff == 0`, or the global pointer is pointing to the `i`th element of an array of `N` elements, where `i` may be equal to `N`, representing a one-past-the-end pointer. At least one of the indices `i-diff` or `i-diff-1` must be a valid element of the same array. A pointer to a non-array object is treated as a pointer to an array of size 1.

52 If `diff == 0`, returns a copy of the global pointer. Otherwise produces a pointer that references the element that is at `diff` positions less than the current element, or a one-past-the-end pointer if the last element of the array is at `diff+1` positions less than the current.

53 `operator-=` modifies the `global_ptr` in-place and returns a reference to this pointer after the operation.

54 These routines are purely a function of their arguments, they are not affected by the context in which they are called.

```
55  template<typename T, memory_kind Kind>
    ptrdiff_t
        global_ptr<T, Kind>::operator-(
                                global_ptr<const T, Kind> rhs) const;
```

56 *Precondition:* `T` must be a complete type. Either `*this == rhs`, or this global pointer is pointing to the `i`th element of an array of `N` elements, and `rhs` is pointing at the `j`th element of the same array. Either pointer may also point one past the end of the array, so that `i` or `j` is equal to `N`. A pointer to a non-array object is treated as a pointer to an array of size 1.

57 If `*this == rhs`, results in 0. Otherwise, returns `i-j`.

58 This routine is purely a function of its arguments, it is not affected by the context in which it is called.

59
```cpp
template < typename T, memory_kind Kind >
global_ptr <T, Kind >& global_ptr <T, Kind >:: operator ++();
template < typename T, memory_kind Kind >
global_ptr <T, Kind > global_ptr <T, Kind >:: operator ++( int );
template < typename T, memory_kind Kind >
global_ptr <T, Kind >& global_ptr <T, Kind >:: operator --();
template < typename T, memory_kind Kind >
global_ptr <T, Kind > global_ptr <T, Kind >:: operator --( int );
```

60 *Precondition:* `T` must be a complete type. In the first two variants, the global pointer must be pointing to an element of an array or to a non-array object. In the third and fourth variants, the global pointer must either be pointing to the `i`th element of an array, where `i >= 1`, or one element past the end of an array or a non-array object.

61 Modifies this pointer to have the value `*this + 1` in the first two variants and `*this - 1` in the third and fourth variants.

   The first and third variants return a reference to this pointer. The second and fourth variants return a copy of the original pointer.

62 This routine is purely a function of its instance, it is not affected by the context in which it is called.

63
```cpp
template < typename T, memory_kind Kind >
bool global_ptr <T, Kind >:: operator ==(
                          global_ptr < const T, Kind > rhs) const ;
template < typename T, memory_kind Kind >
bool global_ptr <T, Kind >:: operator !=(
                          global_ptr < const T, Kind > rhs) const ;
template < typename T, memory_kind Kind >
bool global_ptr <T, Kind >:: operator <(
                          global_ptr < const T, Kind > rhs) const ;
template < typename T, memory_kind Kind >
bool global_ptr <T, Kind >:: operator <=(
                          global_ptr < const T, Kind > rhs) const ;
template < typename T, memory_kind Kind >
bool global_ptr <T, Kind >:: operator >(
                          global_ptr < const T, Kind > rhs) const ;
template < typename T, memory_kind Kind >
bool global_ptr <T, Kind >:: operator >=(
                          global_ptr < const T, Kind > rhs) const ;
```

64   Returns the result of comparing two global pointers. Two global pointers compare equal if they both represent null pointers, or if they represent the same memory address with affinity to the same process. All other global pointers compare unequal.

65   If `Kind == memory_kind::any`, then two non-null global pointers compare equal only if the memory locations they reference have affinity to the same process and represent the same memory address on the same device.

66   A pointer to a non-array object is treated as a pointer to an array of size one. If two global pointers point to different elements of the same array, or to subobjects of two different elements of the same array, then the pointer to the element at the higher index compares greater than the pointer to the element at the lower index. If one pointer points to an element of an array or to a subobject of an element of an array, and the other pointer points one past the end of the array, then the latter compares greater than the former.

67   If global pointers `p` and `q` compare equal, then `p == q`, `p <= q`, and `p >= q` all result in true while `p != q`, `p < q`, and `p > q` all result in false. If `p` and `q` do not compare equal, then `p != q` is true while `p == q` is false.

68   If `p` compares greater than `q`, then `p > q`, `p >= q`, `q < p`, and `q <= p` all result in true while `p < q`, `p <= q`, `q > p`, and `q >= p` all result in false.

69   All other comparisons result in an unspecified value.

70   These routines are purely functions of their arguments, they are not affected by the context in which they are called.

```
71 namespace std {
     template<typename T, memory_kind Kind>
     struct less<global_ptr<T, Kind>>;
     template<typename T, memory_kind Kind>
     struct less_equal<global_ptr<T, Kind>>;
     template<typename T, memory_kind Kind>
     struct greater<global_ptr<T, Kind>>;
     template<typename T, memory_kind Kind>
     struct greater_equal<global_ptr<T, Kind>>;
     template<typename T, memory_kind Kind>
     struct hash<global_ptr<T, Kind>>;
   }
```

December 15, 2023 – LBNL-2001561

72   Specializations of STL function objects for performing comparisons and computing hash values on global pointers. The specializations of `std::less`, `std::less_equal`, `std::greater`, and `std::greater_equal` all produce a strict total order over global pointers, even if the comparison operators do not. This strict total order is consistent with the partial order defined by the comparison operators.

73
```
template < typename T, memory_kind Kind >
std :: ostream& operator <<( std :: ostream &os ,
                             global_ptr <T, Kind > ptr );
```

74   Inserts an unspecified character representation of `ptr` into the output stream `os`. The textual representation of two objects of type `global_ptr<T, Kind>` is identical if and only if the two global pointers compare equal.

75
```
std :: ostream& operator <<( std :: ostream &os , memory_kind k );
```

76   Inserts an unspecified character representation of `k` into the output stream `os`. The textual representation of two values of type `memory_kind` is identical if and only if the two values are equal.

77
```
template < typename T, typename U, memory_kind Kind >
global_ptr <T, Kind >
    static_pointer_cast ( global_ptr <U, Kind > ptr );
template < typename T, typename U, memory_kind Kind >
global_ptr <T, Kind >
    reinterpret_pointer_cast ( global_ptr <U, Kind > ptr );
template < typename T, typename U, memory_kind Kind >
global_ptr <T, Kind >
    const_pointer_cast ( global_ptr <U, Kind > ptr );
```

78   *Precondition:* The expression `static_cast<T*>((U*)nullptr)` must be well-formed for the first variant; `reinterpret_cast<T*>((U*)nullptr)` must be well-formed for the second variant; `const_cast<T*>((U*)nullptr)` must be well-formed for the third variant.

79   Constructs a global pointer whose underlying raw pointer is obtained by using a cast expression on that of `ptr`. The affinity of the result is the same as that of `ptr`.

80   If `rp` is the raw pointer of `ptr`, then the raw pointer of the result is constructed by `static_cast<T*>(rp)` for the first variant, `reinterpret_cast<T*>(rp)` for the second variant, and `const_cast<T*>(rp)` for the third.

```
81  template < memory_kind ToKind , typename T , memory_kind FromKind >
    global_ptr <T , ToKind >
        static_kind_cast ( global_ptr <T , FromKind > ptr );
    template < memory_kind ToKind , typename T , memory_kind FromKind >
    global_ptr <T , ToKind >
        dynamic_kind_cast ( global_ptr <T , FromKind > ptr );
```

82   *Precondition:* `ptr.is_null() || ToKind == memory_kind::any ||`
      `ptr.dynamic_kind() == ToKind` for the first variant

83   Constructs a global pointer with kind `ToKind` from an existing global pointer
      with kind `FromKind`. It is an error if `ToKind != FromKind` and neither `ToKind`
      nor `FromKind` is `memory_kind::any`.

84   In the second variant, the result is a null pointer if
      `ptr.dynamic_kind() != ToKind` and `ToKind != memory_kind::any`.

```
85  // Macro: function template syntax used for clarity
    template < typename T ,  memory_kind Kind >
    global_ptr < MType , Kind > upcxx_memberof ( global_ptr <T , Kind > ptr ,
                                                member - designator MEMBER )
```

86   *Precondition:* `T` must be a complete type. `ptr` is a pointer to a (possibly
      uninitialized) object of type `T`. `T` must be a standard-layout type. `MEMBER` is a
      member designator such that the expression `offsetof(T, MEMBER)` (using the
      standard library macro from `<cstddef>`) is well-formed and valid in the calling
      context.

87   Evaluates to a global pointer referencing the specified member of the object
      referenced by `ptr`. If `MEMBER` specifies a member object with array type, then
      this invocation evaluates to a global pointer referencing the first element of the
      array rather than the array itself.

88   The type parameter `MType` of the returned global pointer preserves constness
      in the same manner as access to `MEMBER` through a raw pointer of type `T*`.
      Thus, if the expression `std::addressof(std::declval<T*>()->MEMBER)` has
      the type `U*`, where `U` is not an array type, then `MType` is the same as `U`. If `U` is
      an array type `W[Dim1]...[DimN]`, then `MType` is `W`.

89   The result of applying the `upcxx_memberof` macro to a static data member or
      a function member is undefined.

```
90  // Macro: function template syntax used for clarity
    template<typename T,  memory_kind Kind>
    future<global_ptr<MType, Kind> >
        upcxx_memberof_general(global_ptr<T, Kind> ptr,
                               member-designator MEMBER)
```

91   *Precondition:* `T` must be a complete type. `ptr` is a pointer to an object of type
      `T`. `MEMBER` is a member designator such that given a valid `T* lp` referencing
      the target object, the expression `lp->MEMBER` is well-formed and valid in the
      calling context. The expression `lp->MEMBER` must not have reference type.

92   Computes a global pointer referencing the specified member of the object ref-
      erenced by `ptr`, using the most efficient mechanism available, and evaluates to
      a future encapsulating that pointer. If the result is determined using purely
      local information, then the progress level is `none` and the result is a readied
      future. Otherwise, the progress level is `internal` and the resulting future will
      be readied during a subsequent user-level progress for the calling persona.

93   If `MEMBER` specifies a member object with array type, then this invocation eval-
      uates to a global pointer referencing the first element of the array rather than
      the array itself.

94   The type parameter `MType` of the global pointer encapsulated in the returned
      future is as described in the specification of `upcxx_memberof`.

95   The result of applying the `upcxx_memberof_general` macro to a static data
      member or a function member is undefined.

96   *Advice to users:* The preconditions of this macro may prohibit applying it
      to objects residing in a device segment whose type hierarchy includes virtual
      base classes, due to device-specific restrictions on host-driven access to vtable
      pointers for such objects.

97   *Advice to users:* The preconditions of this macro prohibit `MEMBER` from speci-
      fying an array element. Instead, a pointer to an array element `ARRAY[idx]` can
      be constructed by using `upcxx_memberof_general` to access the array itself,
      producing a pointer to the base element of the array, and then adding the offset
      of the desired element:

```
upcxx_memberof_general(ptr, ARRAY).then(
  [=](global_ptr<ElementType> gp) { return gp + idx; })
```

98   *UPC++ progress level:* `none` or `internal`

Chapter 4

Storage Management

4.1 Overview

1   `UPC++` provides several flavors of storage allocation involving the shared segment:

2   • The pair of functions `new_` and `delete_` respectively allocate and deallocate space for one object with dynamic storage duration in the shared segment of the calling process and respectively invoke the object constructor/destructor. These are the shared segment analog of C++'s traditional `new` and `delete` operators for non-array types.

3   • The functions `new_array` and `delete_array` respectively allocate and deallocate space for a typed array of objects with dynamic storage duration in the shared segment of the calling process. Array elements allocated by `new_array` are default initialized, and `delete_array` invokes destructors on the elements. These are the shared segment analog of C++'s traditional `new` and `delete` operators for array types.

4   • The functions `allocate` and `deallocate` allocate and deallocate memory with dynamic storage duraction from the shared segment of the calling process, but do not initialize the memory or invoke C++ constructors/destructors. Callers are responsible for initializing the memory and invoking any constructors (for example, via placement new) and destructors. These are the shared segment analog of `std::malloc` and `std::free`.

5   In addition, Chapter 16 describes mechanisms for managing device-resident shared memory segments and shared objects in the memory of accelerators such as GPUs.

4.2   API Reference

```
1  class bad_shared_alloc : public std::bad_alloc;
```

2   An exception type derived from `std::bad_alloc` that is thrown by some shared
       heap allocation functions to indicate failure to allocate shared storage.

```
3  template<typename T, typename ...Args>
   global_ptr<T> new_(Args &&...args);
```

4   *Precondition:* `T(std::forward<Args>(args)...)` must be a valid call to a
       constructor for `T`. `T` must not be an array type.

5   Allocates space for an object of type `T` from the shared segment of the
       calling process. If the allocation succeeds, returns a pointer to the start
       of the allocated memory, and the object is initialized by invoking the con-
       structor `T(std::forward<Args>(args)...)`. If the allocation fails, throws
       `upcxx::bad_shared_alloc`.

6   *Exceptions:* May throw `upcxx::bad_shared_alloc` or any exception thrown
       by the call `T(std::forward<Args>(args)...)`.

```
7  template<typename T, typename ...Args>
   global_ptr<T> new_(const std::nothrow_t &tag, Args &&...args);
```

8   *Precondition:* `T(std::forward<Args>(args)...)` must be a valid call to a
       constructor for `T`. `T` must not be an array type.

9   Allocates space for an object of type `T` from the shared segment of the call-
       ing process. If the allocation succeeds, returns a pointer to the start of
       the allocated memory, and the object is initialized by invoking the construc-
       tor `T(std::forward<Args>(args)...)`. If the allocation fails, returns a null
       pointer.

10   *Exceptions:* May throw any exception thrown by the call
       `T(std::forward<Args>(args)...)`.

11   `template<typename T>`
      `global_ptr<T> new_array(size_t n);`

12   *Precondition:* `T` must be DefaultConstructible. `T` must not be an array type.

13   Allocates space for an array of `n` objects of type `T` from the shared segment of the calling process. If the allocation succeeds, returns a pointer to the start of the allocated memory, and the objects are default initialized[1]. If the allocation fails, throws `upcxx::bad_shared_alloc`.

14   *Exceptions:* May throw `upcxx::bad_shared_alloc` or any exception thrown by the call `T()`. If an exception is thrown by the constructor for `T`, then previously initialized elements are destroyed in reverse order of construction.

15   `template<typename T>`
      `global_ptr<T> new_array(size_t n, const std::nothrow_t &tag);`

16   *Precondition:* `T` must be DefaultConstructible. `T` must not be an array type.

17   Allocates space for an array of `n` objects of type `T` from the shared segment of the calling process. If the allocation succeeds, returns a pointer to the start of the allocated memory, and the objects are default initialized. If the allocation fails, returns a null pointer.

18   *Exceptions:* May throw any exception thrown by the call `T()`. If an exception is thrown by the constructor for `T`, then previously initialized elements are destroyed in reverse order of construction.

19   `template<typename T>`
      `void delete_(global_ptr<T> g);`

20   *Precondition:* `T` must be Destructible. `g` must be either a null pointer or a non-deallocated pointer that resulted from a call to `new_<T, Args...>` on the calling process, for some value of `Args...`.

21   If `g` is not a null pointer, invokes the destructor on the given object and deallocates the storage allocated to it. Does nothing if `g` is a null pointer.

22   *Exceptions:* May throw any exception thrown by the the destructor for `T`.

---

[1]This behavior is analogous to C++ operator `new` for array types, and should not be be confused with value initialization. Default initialization implies that when `T` is a class type, the elements will have their default constructors invoked, but when `T` is a non-class type, the elements are not initialized and will have indeterminate values.

```
23  template < typename T>
    void delete_array ( global_ptr <T> g );
```

24   *Precondition:* `T` must be Destructible. `g` must be either a null pointer or a non-deallocated pointer that resulted from a call to `new_array<T>` on the calling process.

25   If `g` is not a null pointer, invokes the destructor on each object in the given array and deallocates the storage allocated to it. Does nothing if `g` is a null pointer.

26   *Exceptions:* May throw any exception thrown by the the destructor for `T`.

```
27  void* allocate ( size_t size ,
                     size_t alignment = alignof ( std :: max_align_t ));
```

28   *Precondition:* `alignment` is a valid alignment. `size` must be an integral multiple of `alignment`.

29   Allocates `size` bytes of memory from the shared segment of the calling process, with alignment as specified by `alignment`. If the allocation succeeds, returns a pointer to the start of the allocated memory, and the allocated memory is uninitialized. If the allocation fails, returns a null pointer.

```
30  template < typename T>
    global_ptr <T> allocate ( size_t n = 1 ,
                              size_t alignment = alignof (T ));
```

31   *Precondition:* `alignment` is a valid alignment.

32   Allocates enough space for `n` objects of type `T` from the shared segment of the calling process, with the memory aligned as specified by `alignment`. If the allocation succeeds, returns a pointer to the start of the allocated memory, and the allocated memory is uninitialized. If the allocation fails, returns a null pointer.

```
33  void deallocate ( void* p );
```

34   *Precondition:* `p` must be either a null pointer or a non-deallocated pointer that resulted from a call to the first form of `allocate` on the calling process.

35   Deallocates the storage previously allocated by a call to `allocate`. Does nothing if `p` is a null pointer.

```
36  template<typename T>
    void deallocate(global_ptr<T> g);
```

37   *Precondition:* g must be either a null pointer or a non-deallocated pointer that resulted from a call to `allocate<T, alignment>` on the calling process, for some value of `alignment`.

38   Deallocates the storage previously allocated by a call to `allocate`. Does nothing if `g` is a null pointer. Does not invoke the destructor for `T`.

```
39  std::int64_t shared_segment_size();
```

40   Requests a snapshot of the total size of the shared segment for the calling process.

41   Implementations are permitted to return unspecified negative values, which indicate the query is unsupported or encountered some other error.

42   Otherwise, a positive return value indicates the current total size, in bytes, of the shared segment for the calling process. This total size reflects an upper bound on the sum of space consumed by all shared objects allocated but not yet deallocated by the calling process, space available for servicing subsequent such requests, and unspecified implementation overheads (including but not limited to, padding around allocated objects and objects allocated by the implementation).

43   Return values may vary across processes or across calls on a given process in unspecified ways.

```
44  std::int64_t shared_segment_used();
```

45   Requests a snapshot of the occupied size of the shared segment for the calling process.

46   Implementations are permitted to return unspecified negative values, which indicate the query is unsupported or encountered some other error.

47   Otherwise, a positive return value indicates the current occupied size, in bytes, of the shared segment for the calling process. This occupied size reflects an upper bound on the sum of space consumed by all shared objects allocated but not yet deallocated by the calling process and unspecified implementation overheads (including but not limited to, padding around allocated objects and objects allocated by the implementation).

48   Return values may vary across processes or across calls on a given process in unspecified ways.

Chapter 5

Futures and Promises

5.1 Overview

1 In `UPC++`, the primary mechanisms by which a programmer interacts with non-blocking operations are futures and promises.[1] These two mechanisms, usually bound together under the umbrella concept of *futures*, are present in the `C++11` standard. However, while we borrow some of the high-level concepts of `C++`'s futures, many of the semantics of `upcxx::future` and `upcxx::promise` differ from those of `std::future` and `std::promise`. In particular, while futures in `C++` facilitate communicating between threads, the intent of `UPC++` futures is solely to provide an interface for managing and composing non-blocking operations, and they cannot be used directly to communicate between threads or processes.

2 A non-blocking operation is associated with a state that encapsulates both the status of the operation as well as any result values. Each such operation has associated *promise* objects, which can either be explicitly created by the user or implicitly by the runtime when a non-blocking operation is invoked. A promise represents the producer side of the operation, and it is through a promise that the results of the operation are supplied and its dependencies fulfilled. A *future* is the interface through which the status of the operation can be queried and the results retrieved, and multiple future and promise objects may be associated with the same underlying operation. A future thus represents the consumer side of a non-blocking operation.

---

[1]Another mechanism, persona-targeted callbacks, is discussed in §10.4.

5.2 The Basics of Asynchronous Communication

1 A programmer can invoke a non-blocking operation to be serviced by another process, such as a one-sided get operation (Ch. 8) or a remote procedure call (Ch. 9). Such an operation creates an implicit promise and returns an associated future object to the user. When the operation completes, the future becomes ready, and it can be used to access the results. The following demonstrates an example using a remote get (see Ch. 10 on how to make progress with `UPC++`):

```
1 global_ptr<double> ptr = /* obtain some remote pointer */;
2 future<double> fut = rget(ptr);         // initiate a remote get
3 // ...call into upcxx::progress() elided...
4 if (fut.is_ready()) {                    // check for readiness
5   double value = fut.result();           // retrieve result
6   std::cout << "got: " << value << '\n'; // use result
7 }
```

2 In general, a non-blocking operation will not complete immediately, so if a user needs to wait on the readiness of a future, they must do so in a loop. To facilitate this, we provide the `wait` member function, which waits on a future to complete while ensuring that sufficient progress (Ch. 10) is made on internal and user-level state:

```
1 global_ptr<double> ptr = /* obtain some remote pointer */;
2 future<double> fut = rget(ptr);         // initiate a remote get
3 double value = fut.wait();               // wait for completion and
4                                          //  retrieve result
5 std::cout << "got: " << value << '\n';   // use result
```

3 An alternative to waiting for completion of a future is to attach a *callback* or *completion handler* to the future, to be executed when the future completes. This callback can be any function object, including lambda (anonymous) functions, that can be called on the results of the future, and is attached using `then`.

```
1 global_ptr<double> ptr = /* obtain some remote pointer */;
2 auto fut =
3 rget(ptr).then(   // initiate a remote get and register a callback
4   // lambda callback function
5   [](double value) {
6     std::cout << "got: " << value << '\n';   // use result
7   }
8 );
```

4   The return value of `then` is another future representing the results of the callback, if any. This permits the specification of a sequence of operations, each of which depends on the results of the previous one.

5   A future can also represent the completion of a combination of several non-blocking operations. Unlike the standard C++ future, `upcxx::future` is a variadic template, encapsulating an arbitrary number of result values that can come from different operations. The following example constructs a future that represents the results of two existing futures:

```
1 future<double> fut1 = /* one future */;
2 future<int> fut2 = /* another future */;
3 future<double, int> combined = when_all(fut1, fut2);
```

6   Here, `combined` represents the state and results of two futures, and it will be ready when both `fut1` and `fut2` are ready. The results of `combined` are a `std::tuple` whose components are the results of the source futures.

5.3   Working with Promises

1   In addition to the implicit promises created by non-blocking operations, a user may explicitly create a promise object, obtain associated future objects, and then register non-blocking operations on the promise. This is useful in several cases, such as when a future is required before a non-blocking operation can be initiated, or where a single promise is used to count dependencies.

2   A promise can also be used to count *anonymous dependencies*, keeping track of operations that complete without producing a value. Upon creation, a promise has a dependency count of one, representing the unfulfilled results or, if there are none, an anonymous dependency. Further anonymous dependencies can then be registered on the promise. When registration is complete, the original dependency can then be fulfilled to signal the end of registration. The following example keeps track of several remote put operations with a single promise:

```
1 global_ptr<int> ptrs[10] = /* some remote pointers */;
2 // create a promise with no results, dependency count starts at one
3 promise<> prom;
4
5 // do 10 puts, registering each of them on the promise
6 for (int k = 0; k < 10; k++) {
7   // rput implicitly registers itself on the given promise
8   rput(k, ptrs[k], operation_cx::as_promise(prom));
9 }
```

```
10
11 // fulfill initial anonymous dependency, since registration is done
12 future<> fut = prom.finalize();
13
14 // wait for the rput operations to complete
15 fut.wait();
```

5.4 Advanced Callbacks

1 Polling for completion of a future allows simple overlap of communication and computation operations. However, it introduces the need for synchronization, and this requirement can diminish the benefits of overlap. To this end, many programs can benefit from the use of callbacks. Callbacks avoid the need for an explicit wait and enable reactive control flow: future completion triggers a callback. Callbacks allow operations to occur as soon as they are capable of executing, rather than artificially waiting for an unrelated operation to complete before being initiated.

2 Futures are the core abstraction for obtaining asynchronous results, and an API that supports asynchronous behavior can work with futures rather than values directly. Such an API can also work with immediately available values by having the caller wrap the values into a ready future using the `make_future` function template, as in this example that creates a future for an ordered pair of a `double` and an `int`:

```
1 void consume(future<int, double> fut);
2 consume(make_future(3, 4.1));
```

3 Given a future, we can attach a callback to be executed at some subsequent point when the future is ready using the `then` member function:

```
1 future<int, double> source = /* obtain a future */;
2 future<double> result = source.then(
3   [](int x, double y) {
4     return x + y;
5   }
6 );
```

4 In this example, `source` is a future representing an `int` and a `double` value. The argument of the call to `then` must be a function object that can be called on these values. Here, we use a lambda function that takes in an `int` and a `double`. The call to `then` returns a future that represents the result of calling the argument of `then` on the values contained

in `source`. Since the lambda function above returns a `double`, the result of `then` is a `future<double>` that will hold the double's value when it is ready.

5 Instead of returning a plain value, the callback function passed to `then` may directly return a future. In this case, the future returned by `then` has the same type as the future returned by the callback, and both futures represent the same set of results. The future returned by `then` will be readied when two conditions are met: the invocation of the callback function has completed, and the future returned by the callback has itself been readied.

```
1 future <int , double > source = /* obtain a future */;
2 future <double > result = source.then(
3   [](int x, double y) {
4     // return a future <double > that is ready
5     return make_future(x + y);
6   }
7 );
8 // result may not be ready , since the callback will not be executed
9 // until source is ready
```

6 In the example above, the future returned by `then` is readied as soon as the callback completes its execution, since the callback returns a ready future. The future returned by `then` encapsulates the values passed to `make_future`.

7 A callback may also initiate new asynchronous work and return a future representing the completion of that work:

```
1 global_ptr <int > remote_array = /* some remote array */;
2
3 // retrieve remote_array[0]
4 future <int > elt0 = rget(remote_array);
5
6 // retrieve remote_array[remote_array[0]]
7 future <int > elt_indirect = elt0.then(
8   [=](int index) {
9     return rget(remote_array + index);
10   }
11 );
```

8 In this example, a callback is chained onto the result of the first call to `rget`. The future returned by the callback only becomes ready after the operation initiated by the `rget` contained within the callback completes. Thus, `elt_indirect` will be made ready after all of the following:

December 15, 2023 – LBNL-2001561

9   • The operation initiated by the first `rget` completes, producing the value to be passed to the callback.

10 • The invocation of the callback completes, returning a future that represents the eventual results of the second `rget`.

11 • The operation initiated by the second `rget` completes, producing the final `int` value.

12 The final `int` value is the result encapsulated by `elt_indirect`. This example demonstrates how the UPC++ programmer can chain the results of one asynchronous operation into the inputs of the next, to arbitrary degree of nesting.

13 The `then` member function is a combinator for constructing pipelines of transformations over futures. Given a future and a function that transforms that future's value into another value, `then` produces a future representing the transformed value. For example, we can transform, via a future, the value of `elt_indirect` above as follows:

```
future<int> elt_indirect_squared = elt_indirect.then(
  [](int value) {
    return value * value;
  }
);
```

14 As the examples above demonstrate, the `then` member function allows a callback to depend on the result of another future. A more general pattern is for an operation to depend on the results of multiple futures. The `when_all` function template enables this by constructing a single future that combines the results of multiple futures. We can then register a callback on the combined future:

```
future<int> value1 = /* ... */;
future<double> value2 = /* ... */;

future<int, double> combined = when_all(value1, value2);
future<double> result = combined.then(
  [](int x, double y) {
    return x + y;
  }
);
```

15 In the more general case, we may need to combine heterogeneous mixtures of future and non-future types. The `when_all` function template also permits non-future values to be passed in as arguments. Thus, we can use `when_all` to construct a single future that represents the combination of both future and non-future values:

```
1 future<int> value1 = /* ... */;
2 double      value2 = /* ... */;
3
4 future<int, double> combined = when_all(value1, value2);
5 future<double> result = combined.then(
6   [](int x, double y) {
7     return x + y;
8   }
9 );
```

16 The results of a ready future can be obtained as a `std::tuple` using the `result_tuple`
    member function. Individual components can be retrieved by value with the `result` mem-
    ber function template. Unlike with `std::get`, it is not a compile-time error to use an
    invalid index with `result`; instead, the return type is `void` for an invalid index (other than
    -1, which has special handling as described in the API reference below). This simplifies
    writing generic functions on futures, such as the following definition of `wait`:

```
1 template<typename ...T>
2 template<int I=-1>
3 auto future<T...>::wait() {   // C++14-style decl for brevity
4   while (!is_ready()) {
5     progress();
6   }
7   return result<I>();
8 }
```

5.5   Execution Model

1 While some software frameworks provide thread-level parallelism by considering each call-
   back to be a task that can be run in an arbitrary worker thread, this is not the case in
   `UPC++`. In order to maximize performance, our approach to futures is purposefully am-
   bivalent to issues of concurrency. A `UPC++` implementation is allowed to take action as if
   the current thread is the only one that needs to be accounted for. This restriction gives
   rise to a natural execution policy: callbacks registered against futures are always executed
   as soon as possible by the thread that discovers them. There are exactly two scenarios in
   which this may happen:

   1. When a promise is fulfilled.

   2. A callback is registered onto a ready future using the `then` member function.

2 Fulfilling a promise (via `fulfill_result`, `fulfill_anonymous` or `finalize`) is the only operation that can change an associated future from a non-ready to a ready state, enabling callbacks that depend on it to execute. Thus, promise fulfillment is an obvious place for discovering and executing such callbacks. Whenever a thread calls a fulfillment function on a promise, the user must anticipate that any newly available callbacks will be executed by the current thread before the fulfillment call returns.

3 The other place in which a callback will execute immediately is during the invocation of `then` on a future that is already in its ready state. In this case, the callback provided will fire immediately during the call to `then`.

4 There are some common programming contexts where it is not safe for a callback to execute during fulfillment of a promise. For example, it is generally unsafe to execute a callback that modifies a data structure while a thread is traversing the data structure. In such a situation, it is the user's responsibility to ensure that a conflicting callback will not execute. One solution is create a promise that represents a thread reaching its *safe-to-execute* context, and then adding it to the dependency list of any conflicting callback.

```cpp
future<int> value = /* ... */;
// create a promise representing a safe-to-execute state
// dependency count is initially 1
promise<> safe_state;
// create a future that depends on both value and safe_state
future<int> combined = when_all(value, safe_state.get_future());
auto fut = // register a callback on the combined future
combined.then(/* some callback that requires a safe state */);
// do some work, potentially fulfilling value's promise...
// signify a safe state
safe_state.finalize();
// callback can now execute
```

5 As demonstrated above, the user can wait to fulfill the promise until it is safe to execute the callback, which will then allow it to execute.

5.6 Fulfilling Promises

1 As demonstrated previously, promises can be used to both supply values as well as signal completion of events that do not produce a value. As such, a promise is a unified abstraction for tracking the completion of asynchronous operations, whether the operations produce a value or not. A promise represents at most one dependency that produces a value, but it can track any number of anonymous dependencies that do not result in a value.

2   When created, a promise starts with an initial dependency count of 1. For an empty
     promise (`promise<>`), this is necessarily an anonymous dependency, since an empty promise
     does not hold a value. For a non-empty promise, the initial count represents the sole
     dependency that produces a value. Further anonymous dependencies can be explicitly
     registered on a promise with the `require_anonymous` member function:

```
1 promise<int, double> pro; // initial dependency count is 1
2 pro.require_anonymous(10); // dependency count is now 11
```

3   The argument to `require_anonymous` must be nonnegative and the promise's current
     dependency count must be greater than zero, so that a call to `require_anonymous` never
     causes the dependency count to reach zero, which would put the promise in the fulfilled
     state. In the example above, the argument must be greater than -1, and the given argument
     of 10 is valid.

4   Anonymous dependencies can be fulfilled by calling the `fulfill_anonymous` member func-
     tion:

```
1 for (int k = 0; k < 5; k++) {
2   pro.fulfill_anonymous(k);
3 } // dependency count is now 1
```

5   A non-anonymous dependency is fulfilled by calling `fulfill_result` with the produced
     values:

```
1 pro.fulfill_result(3, 4.1); // dependency count is now 0
2 assert(pro.get_future().is_ready());
```

6   Both empty and non-empty promises can be used to track anonymous dependencies. A
     `UPC++` communication operation that operates on a promise *always* increments its depen-
     dency count upon invocation, as if by calling `require_anonymous(1)` on the promise. After
     the operation completes[2], if the completion produces values of type `T...`, then the values
     are supplied to the promise through a call to `fulfill_result`. Otherwise, the completion is
     signaled by fulfilling an anonymous dependency through a call to `fulfill_anonymous(1)`.

7   The rationale for this behavior is to free the user from having to manually increment the
     dependency count before calling an operation on a promise; instead, `UPC++` will implicitly
     perform this increment. This leads to the pattern, shown at the beginning of this chapter,
     of registering operations on a promise and then finalizing the promise to take it out of
     registration mode:

---

[2]See Ch. 7 for details about when the completion notification will occur.

```
1 global_ptr<int> ptrs[10] = /* some remote pointers */;
2 promise<> prom; // dependency count is 1
3
4 for (int i = 0; i < 10; i++) {
5   rput(i, ptrs[i],
6         operation_cx::as_promise(prom)); // register operations
7 }
8
9 future<> fut = prom.finalize(); // fulfill initial dependency
10
11 // wait for the rput operations to complete
12 fut.wait();
```

8 A user familiar with `UPC++` V0.1 will observe that empty promises subsume the capabilities of `event`s in `UPC++` V0.1. In addition, they can take part in all the machinery of promises, futures, and callbacks, providing a much richer set of capabilities than were available in V0.1.

5.7 Lifetime and Thread Safety

1 Understanding the lifetime of objects in the presence of asynchronous control flow can be tricky. Objects must outlive the last callback that references them, which in general does not follow the scoped lifetimes of the call stack. For this reason, `UPC++` automatically manages the state represented by futures and promises, and the state persists for as long as there is a future, promise, or dependent callback that references it. Thus, a user can construct intricate webs of callbacks over futures without worrying about explicitly managing the state representing the callbacks' dependencies or results.

2 Though `UPC++` does not prescribe a specific management strategy, the semantics of futures and promises are analogous to those of standard C++11 smart pointers. As with `std::shared_ptr`, futures and promises may be freely copied, and both the original and the copy represent the same state and are associated with the same underlying set of dependencies and callbacks. Thus, if one copy of a future becomes ready, then so will the other copies, and if a dependency is fulfilled on one copy of a promise, it is fulfilled on the others as well.

3 Given that `UPC++` futures and promises are already thread-unaware to allow the execution strategy to be straightforward and efficient, `UPC++` also makes no thread safety guarantees about internal state management. This enables creation of copies of a future or promise to be a very cheap operation. For example, a future or promise can be captured by value by

a lambda function[3] or passed by value without any performance penalties. On the other hand, the lack of thread safety means that sharing a future or promise between threads must be handled with great caution. Even a simple operation such as making a copy of a future or promise, as when passing it by value to a function, is unsafe if another thread is concurrently accessing an identical future or promise, since the act of copying it can modify the internal management state. Thus, a mutex or other synchronization is required to ensure exclusive access to a future or promise when performing any operation on it.

4   Fulfilling a promise gives rise to an even more stringent demand, since it can set off a cascade of callback execution. Before fulfilling a promise, the user must ensure that the thread has the exclusive right to mutate not just the future associated with the promise, but all other futures that are directly or indirectly dependent on fulfillment of the promise. Thus, when crafting their code, the user must properly manage exclusivity for *islands* of disjoint futures. We say that two futures are in *disjoint islands* if there is no dependency, direct or indirect, between them.

5   A reader having previous experience with futures will note that `UPC++`'s formulation is a significant departure from many other software packages. Futures are commonly used to pass data between threads, like a channel that a producing thread can supply a value into, notifying a consuming thread of its availability. `UPC++`, however, is intended for high-performance computing, and supporting concurrently shareable futures would require synchronization that would significantly degrade performance. As such, futures in `UPC++` are not intended to *directly* facilitate communication between threads. Rather, they are designed for a single thread to manage the non-determinism of reacting to the events delivered by concurrently executing agents, be they other threads or the network hardware.

5.8   API Reference

1   In this and subsequent API-reference sections, `FType` denotes a future type, and `EType` denotes a non-future type derived from the element types of a future. The actual types denoted by `FType` and `EType` differ between uses and are explained in the API descriptions in which they are used.

---

[3]Futures and promises are not TriviallySerializable (or even Serializable) (Ch. 6), so they cannot be captured by copy in a lambda expression that is passed to a remote procedure call (Ch. 9) or an RPC completion (Ch. 7).

5.8.1 future

```
1  template < typename ...T >
   class future ;
```

2   *C++ Concepts:* DefaultConstructible, CopyConstructible, CopyAssignable, Destructible

3   The types in `T` must be complete types (and not `void`). Several member functions impose stricter requirements on `T`.

4   *The constructors, assignment operators, and destructor may be called when* `UPC++` *is in the uninitialized state.*

```
5  template < typename ...T >
   future <T...>::future ();
```

6   Constructs a future that will never become ready.

7   *This function may be called when* `UPC++` *is in the uninitialized state.*

```
8  template < typename ...T >
   future <T...>::~future ();
```

9   Destructs this future object.

10   *This function may be called when* `UPC++` *is in the uninitialized state.*

```
11 template < typename ...T >
   future <T...> make_future (T ...results );
```

12   *Precondition:* Each component of `T` must be MoveConstructible.

13   Constructs a trivially ready future from the given values.

14   *This function may be called when* `UPC++` *is in the uninitialized state.*

```
15 template < typename ...T >
   bool future <T...>::is_ready () const ;
```

16   Returns true if the future is in the ready state, meaning that all its dependencies have been fulfilled and the encapsulated values, if any, are available for consumption.

17
```
template < typename ...T >
bool future <T... >:: ready () const ;
```

18 Deprecated function, equivalent to `this->is_ready()`.

19
```
template < typename ...T >
std :: tuple <T... > future <T... >:: result_tuple () const ;
```

20 *Precondition:* `this->is_ready()`. Each component of `T` must be CopyConstructible.

21 Retrieves the tuple of result values for this future.

22
```
template < typename ...T >
template < int I= -1 >
EType future <T... >:: result () const ;
```

23 *Precondition:* `this->is_ready()`. If `EType` is non-`void`, each component of `EType` must be CopyConstructible.

24 If `I` is in the range `[0, sizeof...(T))`, retrieves the Ith component from the future's results tuple. The return type `EType` is the Ith component of `T`.

25 If `I` is -1, returns the following:

26 - `void` if `T` is empty

27 - if `T` has one element, the single component of the future's results tuple; the return type is `T`

28 - if `T` has multiple elements, the tuple of result values for the future; the return type is `std::tuple<T...>`

29 The return type is `void` if `I` is outside the range `[-1, sizeof...(T))`.

30
```
template < typename ...T >
template < int I= -1 >
EType future <T... >:: result_reference () const ;
```

31 *Precondition:* `this->is_ready()`

32 If `I` is in the range `[0, sizeof...(T))`, retrieves the Ith component from the future's results tuple as a reference. If the Ith component of `T` has type `U`, the return type of this function is:

33 - `U` if `U` is of reference type

34 - `const U&` if `U` is of non-reference type

35   If `I` is -1, returns the following:

36   • `void` if `T` is empty

37   • if `T` has one element, the single component of the future's results tuple as a reference; if the component has type `U`, the return type of this function is:

38   – `U` if `U` is of reference type

39   – `const U&` if `U` is of non-reference type

40   • if `T` has multiple elements, the tuple of result values for the future as references; the return type is `std::tuple<V...>`, where if the nth component of `T` has type `U`, then the nth component of `V` has type:

41   – `U` if `U` is of reference type

42   – `const U&` if `U` is of non-reference type

43   The return type is `void` if `I` is outside the range `[-1, sizeof...(T))`.

44 ```
template<typename ...T>
std::tuple<T...> future<T...>::wait_tuple() const;
```

45    *Precondition:* Each component of `T` must be CopyConstructible.

46    Blocks until the future is ready, while making `UPC++` user-level progress. See Ch. 10 for a discussion of progress. The return value is the same as that produced by calling `result_tuple()` on the future.

47    *This function may not be invoked from the restricted context (§10.2).*

48    *UPC++ progress level:* `user`

49 ```
template<typename ...T>
template<int I=-1>
EType future<T...>::wait() const;
```

50   *Precondition:* If `EType` is non-`void`, each component of `EType` must be CopyConstructible.

51   Blocks until the future is ready, while making `UPC++` user-level progress. See Ch. 10 for a discussion of progress. The return value is the same as that produced by calling `result<I>()` on the future.

52   *This function may not be invoked from the restricted context (§10.2).*

53   *UPC++ progress level:* `user`

54 ```
    template < typename ...T >
    template < int I = -1 >
    EType future <T... >:: wait_reference () const ;
    ```

55    Blocks until the future is ready, while making `UPC++` user-level progress. See
      Ch. 10 for a discussion of progress. The return value is the same as that
      produced by calling `result_reference<I>()` on the future.

56    *This function may not be invoked from the restricted context (§10.2).*

57    *UPC++ progress level:* `user`

58  ```
    template < typename ...T >
    template < typename Func >
    FType future <T... >:: then ( Func && func ) const ;
    ```

      Preconditions:

59   • If `Func&&` is an rvalue-reference type, the underlying decayed type
          (`std::decay<Func&&>::type`) must be MoveConstructible.

60   • If `Func&&` is an lvalue-reference type, the underlying decayed type
          (`std::decay<Func&&>::type`) must be CopyConstructible.

61   • `func` must be invokable as `std::move(func)(...)` on a sequence of
          `sizeof...(T)` arguments, where if the nth component of `T` has type `U`,
          then the nth argument provided to `func` has type:

62   − `U` if `U` is of reference type

63   − `const U&` if `U` is of non-reference type

64   • If the return type `RetType` of `func` is of non-reference or rvalue-reference
          type, the underlying decayed type (`std::decay<RetType>::type`) must
          be MoveConstructible.

65   • The invocation of `func` must not throw an exception.

66   Returns a new future representing the return value of the given function object
      `func` when invoked on the results of this future as its argument list. The return
      type `FType` and semantics are as follows:

67
- If `func` returns a future type `future<U...>`, then `FType` is also `future<U...>`. The future returned by `then` encapsulates the same set of results as the future returned by `func`. The future returned by `then` is readied when both the invocation of `func` has completed and the future returned by that invocation has been readied.

68
- If `func` returns a non-future, non-`void` type `U`, then `FType` is `future<U>`. The future returned by `then` encapsulates the result of `func`, and it is readied upon completion of the invocation of `func`.

69
- If `func` has `void` return type, then `FType` is `future<>`. The future returned by `then` represents whether or not the invocation of `func` has completed, and it is readied upon completion of that invocation.

70 The function object will be invoked in one of two situations:

71
- Immediately before `then` returns if this future is in the ready state.

72
- During a promise fulfillment which would directly or indirectly make this future transition to the ready state.

73
```
template<typename ...T>
future<ETypes...> when_all(T&& ...futures_or_values);
```

74 *Precondition:* For any future type `future<U...>` in T, each argument type `U` must be CopyConstructible. Each non-future type in `T` must be MoveConstructible.

75 Given a variadic list of arguments consisting of futures and non-future values, constructs a future representing the readiness of all arguments. The results tuple of this future will consist of the concatenation of the results or non-future values represented by each argument. The type parameters of the returned object (`ETypes...`) is the ordered concatenation of the following over each component of `T`:

76
- `U...` if the nth component of `T` is a future type or a reference to a future type `future<U...>`

77
- `U` if the nth component of `T` is a non-future type or a reference to a non-future type `U`

If `T...` is empty, then the result is a trivially ready `future<>`.

78 *This function may be called when* `UPC++` *is in the uninitialized state.*

79
```
template<typename T>
future<ETypes...> to_future(T future_or_value);
```

80   *Precondition:* `T` must be MoveConstructible if it is a non-future type.

81   Constructs a future that encapsulates the value represented by `future_or_value`. If `T` is of type `future<U...>`, then `ETypes...` is the same as `U...`, and the returned future is a copy of `future_or_value`. If `T` is not a future, then the call `to_future(arg)` is semantically equivalent to `make_future(arg)`. In this case, `ETypes...` is `T`, and the function returns a ready future whose encapsulated value is `future_or_value`.

82   *This function may be called when* `UPC++` *is in the uninitialized state.*

5.8.2 promise

1
```
template < typename ... T >
class promise ;
```

2 *C++ Concepts:* DefaultConstructible, CopyConstructible, CopyAssignable, Destructible

3 The types in `T` must be complete types (and not `void`). Several member functions impose stricter requirements on `T`.

4
```
template < typename ... T >
promise <T... >:: promise ( std :: intptr_t dependency_count =1);
```

5 *Precondition:* `dependency_count >= 1`

6 Constructs a promise with its results uninitialized and the given initial dependency count. The state of the resulting promise is independent of all existing promises.

7 *This function may be called when* `UPC++` *is in the uninitialized state.*

8
```
template < typename ... T >
promise <T... >:: ~ promise ();
```

9 Destructs this promise object.

10 *This function may be called when* `UPC++` *is in the uninitialized state.*

11
```
template < typename ... T >
void promise <T... >:: require_anonymous ( std :: intptr_t count ) const ;
```

12 *Precondition:* `count` is nonnegative. The dependency count of this promise is greater than 0.

13 Adds `count` to this promise's dependency count.

```
14  template<typename ...T>
    template<typename ...U>
    void promise<T...>::fulfill_result(U &&...results) const;
```

15 *Precondition:* `fulfill_result` has not been called on this promise or a copy of this promise before, and the dependency count of this promise is greater than zero. `T` and `U` must have the same number of components, and each component of `T` must be constructible from the corresponding component of `U` (i.e., `std::is_constructible<std::tuple<T...>, U&&...>::value` must be true).

16 Initializes the promise's result tuple with the given values and decrements the dependency counter by 1. If the dependency counter reaches zero as a result of this call, the associated future is set to ready, and callbacks that are waiting on the future are executed on the calling thread before this function returns.

```
17  template<typename ...T>
    void promise<T...>::fulfill_anonymous(std::intptr_t count) const;
```

18 *Precondition:* `count` is nonnegative. The dependency count of this promise is greater than zero and greater than or equal to `count`. If the dependency count is equal to `count` and `T` is not empty, then the results of this promise must have been previously supplied by a call to `fulfill_result`.

19 Subtracts `count` from the dependency counter. If this produces a zero counter value, the associated future is set to ready, and callbacks that are waiting on the future are executed on the calling thread before this function returns.

```
20  template<typename ...T>
    future<T...> promise<T...>::get_future() const;
```

21 Returns the future representing this promise being fulfilled. Repeated calls to `get_future` return equivalent futures with the guarantee that no additional memory allocation is performed.

```
22  template<typename ...T>
    future<T...> promise<T...>::finalize() const;
```

23 Equivalent to calling `this->fulfill_anonymous(1)` and then returning the result of `this->get_future()`.

Chapter 6

Serialization

1 As a communication library, `UPC++` needs to send C++ objects between processes that might be separated by a network interface. The underlying GASNet-EX [6] networking interface sends and receives bytes, thus, `UPC++` needs to be able to convert C++ objects to and from bytes.

2 `UPC++` communication operations such as remote procedure calls (Ch. 9) *serialize* C++ objects, converting them to raw bytes, before sending the data to the destination. Upon receiving the data at the destination, the library *deserializes* the raw bytes back into C++ objects. We refer to the data channel between the sender and receiver of an operation as a *byte stream*; the sender writes data sequentially to the stream, and the receiver sequentially reads data out of it.

6.1 Serialization Concepts

1 `UPC++` defines the concepts *TriviallySerializable* and *Serializable* that describe what form of serialization a C++ type supports. Figure 6.1 helps summarize the relationship of these concepts.

2 A type `T` is TriviallySerializable if it is semantically valid to copy an object by copying its underlying bytes. `UPC++` serializes such types by making a byte copy.

3 A type `T` is considered TriviallySerializable if either of the following holds:

4 - `T` is TriviallyCopyable (i.e. `std::is_trivially_copyable<T>::value` is `true`), and (if `T` is of class type) `T` does not implement any class serialization interface described in §6.2

51

```
┌──────────────────────────────────────────────────────────────┐
│  Serializable                                                  │
│                             ┌────────────────────────────────┐ │
│  Types implementing any     │ TriviallySerializable          │ │
│  Class Serialization interface │                             │ │
│                             │ TriviallyCopyable              │ │
│  -or-                       │                                │ │
│                             │  -or-                          │ │
│  Some spec-provided types   │ is_trivially_serializable      │ │
│     (e.g., view<T>)         │    specialized to true         │ │
│                             └────────────────────────────────┘ │
└──────────────────────────────────────────────────────────────┘
```
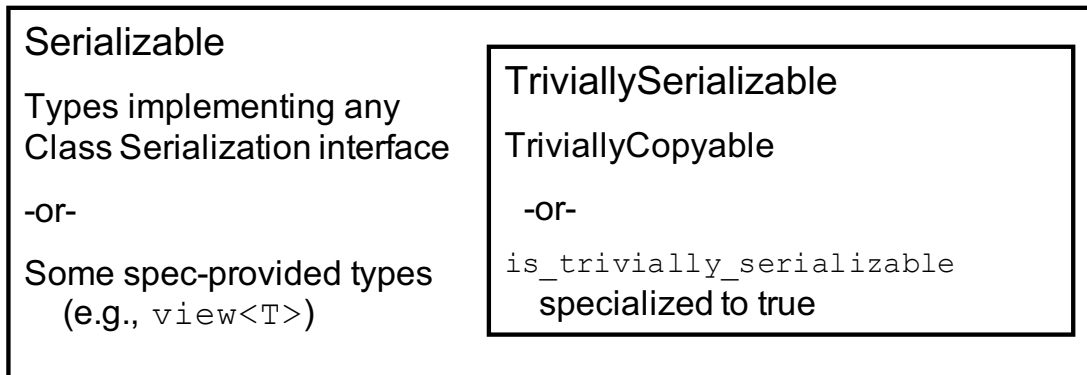
Figure 6.1: Serializable UPC++ concepts type hierarchy.

5   • `upcxx::is_trivially_serializable<T>` is specialized to provide a member constant `value` that is `true`

6   In the latter case, `UPC++` treats the type `T` as if it were TriviallyCopyable for the purposes of serialization. Thus, `UPC++` will serialize an object of type `T` by making a byte copy, and it will assume `T` is TriviallyDestructible when destroying a deserialized object of type `T`.

7   A type `T` is considered Serializable if one of the following holds:

8   • `T` is TriviallySerializable

9   • `T` is of class type and implements a class serialization interface described in §6.2

10   • `T` is explicitly described as Serializable by this specification

11   The type trait `upcxx::is_trivially_serializable<T>` provides a member constant `value` that is `true` if `T` is TriviallySerializable and `false` otherwise. This trait may be specialized for user types (types that are not defined by the C++ or `UPC++` standards).

12   The type trait `upcxx::is_serializable<T>` provides a member constant `value` that is `true` if `T` is Serializable and `false` otherwise. This trait may not be specialized by the user for any types.

13   Several `UPC++` communication operations require that the objects to be transferred are of TriviallySerializable type. The C++ standard allows implementations to determine whether or not lambda function objects are TriviallyCopyable, so whether or not such objects are TriviallySerializable is implementation-dependent.

14   Serializability of a type `T` does not imply that objects of type `T` are meaningful on another process. In particular, C++ pointer-to-object and pointer-to-function types are TriviallySerializable, but it is generally invalid to dereference a local pointer that originated from

another process. More generally, objects that represent local process resources (e.g., file descriptors) are usually not meaningful on other processes, whether their types are Serializable or not.

6.2 Class Serialization Interface

1. For a class `T` that requires nontrivial serialization, `UPC++` provides several different mechanisms for specifying how serialization and deserialization are to be performed.

1. Declare which member variables of `T` to serialize with `UPCXX_SERIALIZED_FIELDS`. `UPC++` automatically generates the required serialization logic.

2. Specify expressions for computing the data to be serialized with `UPCXX_SERIALIZED_VALUES`. `UPC++` automatically generates logic to evaluate the expressions in serialization and invoke a constructor with the resulting values in deserialization.

3. Define a public, nested `T::upcxx_serialization` member type with public `serialize` and `deserialize` member-function templates.

4. Define a specialization of `upcxx::serialization<T>` with public `serialize` and `deserialize` member-function templates.

2. If any of these mechanisms is used on a type `T`, then `T` is Serializable but not TriviallySerializable, unless `upcxx::is_trivially_serializable<T>` is specialized to provide a member constant `value` that is `true`.

3. It is an error if more than one of the first three mechanisms (`UPCXX_SERIALIZED_FIELDS`, `UPCXX_SERIALIZED_VALUES`, or a nested `upcxx_serialization` class) is used directly by the same class `T`.

4. It is an error if both an explicit specialization of `upcxx::serialization<T>` is defined and `upcxx::is_trivially_serializable<T>` is specialized to provide a member constant `value` that is `true`.

5. An explicit specialization of `serialization<T>` or `is_trivially_serializable<T>` takes precedence over the mechanisms that are nested within a class (`UPCXX_SERIALIZED_FIELDS`, `UPCXX_SERIALIZED_VALUES`, or a nested `upcxx_serialization` class).

6.2.1 UPCXX_SERIALIZED_FIELDS

1 If serialization of a type `T` can be accomplished by recursively serializing a fixed subset of its member variables, the variadic `UPCXX_SERIALIZED_FIELDS` macro may be used to declare this subset. `UPC++` will then automatically generate the code to serialize and deserialize objects of type `T`.

2 The following is an example of using `UPCXX_SERIALIZED_FIELDS`:

```
struct UserType {
  U a;
  V b;
  W c;

  UPCXX_SERIALIZED_FIELDS(a, b, c)
};
```

3 The macro `UPCXX_SERIALIZED_FIELDS` must be invoked directly within a class definition in a context that has `public` access level. The macro arguments must name non-static member variables of the class or an application of `UPCXX_SERIALIZED_BASE` (§6.2.5). A bit-field data member may not be used as an argument. In addition, the following must hold:

4 • The class must have a default constructor; the default constructor may have any access level.

5 • Each argument to `UPCXX_SERIALIZED_FIELDS` must be of a non-array type `T`, or a (possibly multidimensional) array of elements of type `T`, where:

6 – `T` must not be qualified with `const`

7 – `T` must be Serializable and Destructible

8 – `T` and `deserialized_type_t<T>` (§6.2.6) must be the same type

9 `UPC++` serializes an object of a type `T` that invokes `UPCXX_SERIALIZED_FIELDS` by serializing each member variable that is an argument to the macro in some unspecified order. Deserialization starts by default constructing an object of type `T`. Then each member variable listed in `UPCXX_SERIALIZED_FIELDS` is destructed and overwritten by an object deserialized from the byte stream, in the same order as serialization. Member variables elided from `UPCXX_SERIALIZED_FIELDS` are not overwritten; they retain their initial values as determined by the default constructor for `T`.

6.2.2 UPCXX_SERIALIZED_VALUES

1 If serialization of a type `T` consists of computing values to be inserted into the byte stream and using those values in deserialization to reconstruct the object, the variadic `UPCXX_SERIALIZED_VALUES` macro may be used. Each argument to the macro must be an expression that can be evaluated from the body of a non-static, `const` member function or an application of `UPCXX_SERIALIZED_BASE` (§6.2.5), and `T` must have a constructor that can be invoked with the resulting rvalues from the body of a static member function. The expressions provided to `UPCXX_SERIALIZED_VALUES` are evaluated in an unspecified order, and the types of the values must be Serializable.

2 The following is an example that uses `UPCXX_SERIALIZED_VALUES` to serialize a type using single-precision rather than the original double-precision floating-point values:

```
1 struct Point {
2   double x;
3   double y;
4
5   Point(float a, float b) : x(a), y(b) {}
6
7   UPCXX_SERIALIZED_VALUES(float(x), float(y))
8 };
```

3 The macro `UPCXX_SERIALIZED_VALUES` must be invoked directly within a class definition in a context that has `public` access level.

6.2.3 Custom Serialization

1 Serialization for a class `T` may be customized by directly writing subobjects to a byte stream and reading them back out in deserialization. The following is an example of specifying custom serialization for a class:

```
1 class UnrolledList {
2   struct Node {
3     int data[NODE_CAPACITY];
4     int node_size;
5     Node* next;
6   };
7
8   Node* first;
9   std::size_t size_;
10
```

```
11    Node* extend (); // allocate a new Node and place at the end
12
13 public:
14    UnrolledList () : first (nullptr), size_ (0) {}
15
16    struct upcxx_serialization {
17      // Write an UnrolledList into the given writer.
18      template < typename Writer >
19      static void serialize (Writer &writer ,
20                             const UnrolledList &obj) {
21        writer.write (obj.size_);
22        for (Node* crnt = obj.first; crnt; crnt = crnt ->next) {
23          writer.write_sequence (crnt ->data ,
24                                 crnt ->data+crnt ->node_size);
25        }
26      }
27
28      // Read an UnrolledList from the given reader into the
29      // provided storage.
30      template < typename Reader , typename Storage >
31      static UnrolledList* deserialize (Reader &reader ,
32                                        Storage storage) {
33        UnrolledList* result = storage.construct ();
34        std:: size_t count = reader.template read <std:: size_t >();
35        result ->size_ = count;
36        for (std:: size_t read_count = 0; read_count < count;
37             read_count += NODE_CAPACITY) {
38          Node* node = result ->extend ();
39          node ->node_size = std::min (count-read_count ,NODE_CAPACITY);
40          reader.template read_sequence_overwrite <int >(
41            node ->data ,
42            node ->node_size
43          );
44        }
45        return result;
46      }
47    };
48 };
```

2 A Writer is an object of an opaque type that provides an interface for writing to a byte stream. Writers provide the following member-function templates:

3 • `write(item)` writes a single object to the byte stream.

4 • `write_sequence(begin, end)` writes a sequence of objects to the byte stream and returns the number of objects in the sequence.

5 • `write_sequence(begin, end, num_items)` provides the same behavior as `write_sequence(begin, end)`, but is more efficient when `begin` and `end` are not RandomAccessIterators.

6 • `reserve<T>()` reserves space in the byte stream for a single object of TriviallySerializable type `T` and returns a handle to the location in the stream. The location is written by calling `commit(handle, object)`, where `object` has type T.

7 The combination of `reserve`, `write_sequence`, and `commit` can be used to write a sequence of unknown length, prefixing the sequence with the actual length:

```
1   // reserve space for the length, to be written later
2   auto handle = writer.template reserve<size_t>();
3   // write the sequence
4   size_t length = writer.write_sequence(begin, end);
5   // write the actual length prior to the sequence
6   writer.commit(handle, length);
```

8 A Reader is an object of an opaque type that provides an interface for reading from a byte stream. Readers provide the following member-function templates:[1]

9 • `read<T>()` reads and returns a single object from the byte stream. `T` must be the type of the object written to the current location in the byte stream, and the returned object is of type `deserialized_type_t<T>`.

10 • `read_overwrite<T>(object)` overwrites the given object with an object read from the byte stream. The call invokes the destructor for `deserialized_type_t<T>` on `object` before overwriting it, and it returns the address of the resulting object.

11 • `read_into<T>(optional_obj)` reads a single object into a `upcxx::optional<deserialized_type_t<T>>`. The `upcxx::optional` is reset before a new object is read into the optional. The return value of the call is the address of the resulting object.

---

[1]Note that due to C++ typechecking rules, invocations of these member-function templates must be explicitly instantiated using a `template` keyword, eg: `reader.template read<int>()`

12   • `read_into<T>(pointer)` reads an object directly into the memory pointed to by `pointer`. Any existing object in the memory denoted by the pointer is not destructed. If `pointer` points to an existing object, `read_overwrite<T>(*pointer)` should be used instead. The return value of the call is the address of the resulting object.

13   • `read_sequence_overwrite<T>(array, num_items)` reads a sequence of `num_items` objects from the byte stream and places them into an array at the memory denoted by `array` after destructing the existing objects located at `array`.

14   • `read_sequence_into<T>(pointer, num_items)` reads a sequence of `num_items` objects from the byte stream and places them into an array at the memory denoted by `pointer`. If `pointer` points to a location with existing objects, `read_sequence_overwrite<T>(pointer, num_items)` should be used instead.

15 `read_sequence_overwrite` and `read_sequence_into` are semantically equivalent to a sequence of calls to `read_overwrite` or `read_into`, respectively, but they may provide better performance.

16 For each of these Reader member-function templates, `T` must be the same type of the original objects written in serialization by `write`, `write_sequence`, or `commit`. However, the objects constructed in deserialization are of type `deserialized_type_t<T>`, which may be different from `T`.

17 Serialization and deserialization for a class `T` may be customized by defining either a public, nested, member type `T::upcxx_serialization` or an explicit specialization of `upcxx::serialization<T>`. The nested class or specialization must define the following public member-function templates:

```
1 template<typename Writer>
2 static void serialize(Writer& writer, T const& object);
3
4 template<typename Reader, typename Storage>
5 static U* deserialize(Reader& reader, Storage storage);
```

18 A Storage is an object of an opaque type that provides the following member-function template:

19   • `construct(args...)`, when invoked on a Storage object that was passed to the `deserialize` member-function template for type `T`, forwards `args...` to the constructor for type `deserialized_type_t<T>` and returns a pointer to the newly constructed object. The `deserialize` member-function template must return this same pointer.

20 For compatibility purposes, `UPC++` allows the following deprecated interface for the `deserialize` template:

```
1 template < typename Reader >
2 static U* deserialize ( Reader& reader , void* pointer );
```

21 `pointer` points to a location with appropriate storage and alignment for an object of type `U`. Definitions of `deserialize` that conform to this interface must use placement `new` to construct the resulting object in the provided location and return a pointer to the object.

22 It is an error if either `T::upcxx_serialization` or an explicit specialization of `upcxx::serialization<T>` is defined without the required public member-function templates. The behavior is unspecified if `T::upcxx_serialization` or `upcxx::serialization<T>` defines both interfaces for the `deserialize` member-function template.

23 `UPC++` invokes `serialize(writer, object)` to serialize `object` into a byte stream, where `object` has type `T`. Similarly, `deserialize(reader, storage)` is invoked to deserialize an object of type `T`. The return type of `deserialize` must be a pointer `U*`, where `U` is the type of the resulting object, which may be distinct from the type `T` passed to serialization.

24 As described in §6.2.6, the types `serialization_traits<T>::deserialized_type` and `deserialized_type_t<T>` are defined as aliases for the type `U`, where `U*` is the return type of `deserialize`.

6.2.4 Restrictions on Class Serialization

1 There are restrictions on which actions serialization/deserialization routines and expressions may perform. The following restrictions apply to constructors and destructors invoked when deserializing an object that uses `UPCXX_SERIALIZED_FIELDS`, expressions passed to `UPCXX_SERIALIZED_VALUES` and the constructor invoked upon deserialization, and all statements executed within any call to the member-function templates `serialize` and `deserialize` (§6.2.3):

    1. Serialization/deserialization may not call any `UPC++` routine with a progress level other than `none`.

    2. If multiple application threads in the same process may concurrently invoke serialization/deserialization, the expressions or routines must be thread-safe and permit concurrent invocation from multiple threads.

2 Serialization/deserialization is only invoked by `UPC++` functions with a progress level of `internal` or `user`. Calls to the member-function templates of a Writer synchronously

invoke serialization on the argument objects, and calls to the member-function templates of a Reader synchronously invoke deserialization to construct the resulting objects.

6.2.5 Serialization and Inheritance

1 Serialization mechanisms that are defined within a class `T` (`UPCXX_SERIALIZED_FIELDS`, `UPCXX_SERIALIZED_VALUES`, or a nested `upcxx_serialization` class) are inherited by the derived classes of `T`. The resulting behavior differs depending on which mechanism is inherited:

2   • If a derived class `U` inherits a nested `upcxx_serialization` class, then serializing an object of type `U` and deserializing produces an object of the deserialized counterpart of base type `T`. Thus, `deserialized_type_t<U>` (§6.2.6) is `deserialized_type_t<T>`.

3   • If a derived class `U` inherits serialization defined using `UPCXX_SERIALIZED_FIELDS` or `UPCXX_SERIALIZED_VALUES`, then serializing an object of type `U` and deserializing produces an object of type `U`. Thus, `deserialized_type_t<U>` (§6.2.6) is `U`.

4 If a derived class `U` inherits serialization defined using `UPCXX_SERIALIZED_FIELDS` or `UPCXX_SERIALIZED_VALUES`, the constructors required by those mechanisms must be members of `U`, and these constructors must additionally have `public` access level.

5 A derived class `U` may provide its own serialization by directly defining one of these mechanisms itself, or by specializing `serialization<U>` or `is_trivially_serializable<U>`.

6 Specializations of `serialization<T>` or `is_trivially_serializable<T>` do not affect serialization of derived classes of `T`.

7 A derived class `U` may disable serialization, when it would otherwise be inherited, by invoking the `UPCXX_SERIALIZED_DELETE` macro. The macro must be invoked directly within the definition of `U` in a context that has `public` access level, and it is subsequently inherited by derived classes of `U`. The following is an example of using `UPCXX_SERIALIZED_DELETE`:

```
1 struct Derived : Base {
2   UPCXX_SERIALIZED_DELETE()
3 };
```

8 `UPC++` serialization does not perform dynamic dispatch. Thus, the call `writer.write(object)` uses the static type of `object` to determine how to serialize `object`, regardless of the actual runtime type of the object.

9 Class serialization for a derived class `U` may explicitly serialize each of its base subobjects:

10   • The expression `UPCXX_SERIALIZED_BASE(B)` may be passed as an argument to `UPCXX_SERIALIZED_FIELDS` to serialize the subobject of base type `B`. `B` must be Serializable and Destructible. `B` must be neither polymorphic nor abstract.

11   Since the order in which the arguments to `UPCXX_SERIALIZED_FIELDS` are serialized and deserialized is unspecified, the behavior is undefined if both `UPCXX_SERIALIZED_BASE(B)` and a member variable inherited from `B` are passed to `UPCXX_SERIALIZED_FIELDS`.

12   • The expression `UPCXX_SERIALIZED_BASE(B)` may be passed as an argument to `UPCXX_SERIALIZED_VALUES` to serialize the subobject of base type `B`. `B` must be Serializable and must not be abstract. The constructor of `U` invoked by deserialization must accept an rvalue of type `B` as the corresponding argument.

13   • Custom serialization may serialize a base subobject by casting the object to a base class and writing the result. The base type must be Serializable. Deserialization requires reading a base-type object into a temporary before passing it to a derived-class constructor. The following is an example:

```cpp
struct Derived : Base {
  X d;
  Derived(Base&& base, X&& x)
    : Base(std::forward(base)), d(std::forward(x)) {}

  struct upcxx_serialization {
    template<typename Writer>
    static void serialize(Writer& writer,
                          Derived const& object) {
      writer.write(static_cast<Base const&>(object));
      writer.write(object.d);
    }

    template<typename Reader, typename Storage>
    static Derived* deserialize(Reader& reader,
                                Storage storage) {
      Base base = reader.template read<Base>();
      X x = reader.template read<X>();
      return storage.construct(std::move(base),
                               std::move(x));
    }
  };
};
```

6.2.6 Serialization Traits

1 As mentioned in §6.2.3, custom `UPC++` deserialization may produce an object of a different type than that of the original serialized object. `UPC++` provides the `serialization_traits` class template that enables a user to determine the type of the deserialized object: `serialization_traits<T>::deserialized_type` is an alias for the type resulting from deserializing an object of type `T`. The top-level alias template `deserialized_type_t` is an alias for `serialization_traits<T>::deserialized_type`.

2 The `serialization_traits` template also provides a static member function that converts an object to its deserialized counterpart. The call `serialization_traits<T>::deserialized_value(object)` returns a value of type `serialization_traits<T>::deserialized_type`. The latter must be Movable, and `object` must not be a `view` (§6.7).

6.3 Standard-Library Containers

1 `UPC++` supports serialization of several standard-library container types.

2 The following fixed-size containers are TriviallySerializable when the element types `T`, `T1` and `T2`, or `T...` are all TriviallySerializable. They are Serializable when the element types are all Serializable:

3 • `std::array<T, N>`

4 • `std::pair<T1, T2>`

5 • `std::tuple<T...>`

6 `UPC++` treats `std::pair<T1, T2>` and `std::tuple<T...>` as TriviallySerializable when `T1`, `T2`, and `T...` are TriviallySerializable even when the C++ implementation does not consider the pair or tuple to be TriviallyCopyable.

7 The following sequence container types are Serializable when the template parameters (`T` and `Allocator`) are all Serializable:

8 • `std::vector<T, Allocator>`

9 • `std::deque<T, Allocator>`

10 • `std::list<T, Allocator>`

11 The following set container types are Serializable when the template parameters (`Key`, `Compare`, `Hash`, `KeyEqual`, and `Allocator`) are all Serializable:

12   • `std::set<Key, Compare, Allocator>`

13   • `std::multiset<Key, Compare, Allocator>`

14   • `std::unordered_set<Key, Hash, KeyEqual, Allocator>`

15   • `std::unordered_multiset<Key, Hash, KeyEqual, Allocator>`

16 The following map container types are Serializable when the template parameters (`Key`, `T`, `Compare`, `Hash`, `KeyEqual`, and `Allocator`) are all Serializable:

17   • `std::map<Key, T, Compare, Allocator>`

18   • `std::multimap<Key, T, Compare, Allocator>`

19   • `std::unordered_map<Key, T, Hash, KeyEqual, Allocator>`

20   • `std::unordered_multimap<Key, T, Hash, KeyEqual, Allocator>`

21 The type `std::basic_string<CharT, Traits, Allocator>` is Serializable when the template parameter `Allocator` is Serializable.

22 The following types are also Serializable:

23   • `std::allocator<T>`

24   • standard specializations of `std::hash<T>`

25 Typical library implementations of `std::equal_to<T>`, `std::not_equal_to<T>`, `std::greater<T>`, `std::less<T>`, `std::greater_equal<T>`, and `std::less_equal<T>` are TriviallyCopyable and thus TriviallySerializable.

26 When serializing a container that has a template parameter of `Compare`, `Hash`, `KeyEqual`, or `Allocator`, UPC++ invokes the corresponding observer member function (e.g., `key_comp()` or `get_allocator()`), serializes the resulting object, and passes the deserialized object as an argument to the constructor when deserializing the container.

27 UPC++ allows the template arguments (e.g. `T` and `Key` in the types above) of containers to produce different types upon deserialization.
For example, the container `std::vector<T, Allocator>` is deserialized as:
`std::vector<deserialized_type_t<T>,deserialized_type_t<Allocator>>`,
and the container `std::map<Key, T, Compare, Allocator>` is deserialized as:
`std::map<deserialized_type_t<Key>, deserialized_type_t<T>,`
    `deserialized_type_t<Compare>, deserialized_type_t<Allocator>>`.

28 When serializing a container, the deserialized types (e.g. `deserialized_type_t<T>` and `deserialized_type_t<Key>`) corresponding to the template type arguments must be Move-Constructible.

6.4   References, Arrays, and CV-Qualified Types

1 A reference type *cq* `T&` or *cq* `T&&`, as well as `std::reference_wrapper<`*cq* `T>`, is Serializable when `T` is Serializable. An object of such type is serialized by serializing the referent, and deserialization produces an object of non-reference type `deserialized_type_t<T>`. Thus, `deserialized_type_t<`*cq* `T&>`, `deserialized_type_t<`*cq* `T&&>`, and `deserialized_type_t<std::reference_wrapper<`*cq* `T>>` are all the same as `deserialized_type_t<T>`, without any top-level `const` qualifier.

2 A reference or reference-wrapper type is never TriviallySerializable.

3 An array type is never Serializable. However, an array may be passed to `UPCXX_SERIALIZED_FIELDS` or the `write` member-function template of a Writer if the element type of the array is Serializable. Such an array type may also be used with the `read_into` member-function template of a Reader to read an entire array into a given memory location.

4 The type `T const` is Serializable when `T` is Serializable, and it is TriviallySerializable when `T` is TriviallySerializable. Deserialization preserves the original `const` qualifier for non-reference types. Thus, an RPC (§9) of a function whose return type is `T const` by default produces a future (§5) whose type is `future<deserialized_type_t<T> const>`. Similarly, deserialization of a `std::pair<T1 const, T2 const>` produces an object of type `std::pair<deserialized_type_t<T1> const, deserialized_type_t<T2> const>`. On the other hand, due to the rules for deserializing references above, deserialization of a `std::pair<T1 const&, T2 const>` produces an object of type `std::pair<deserialized_type_t<T1>, deserialized_type_t<T2> const>`.

5 The types `T volatile` and `T const volatile` are not Serializable.

6 It is an error to define an explicit specialization `is_trivially_serializable<T>` or `serialization<T>` if `T` is a reference, array, or cv-qualified type.

6.5 Functions

1 In Chapter 7 (*Completion*) and Chapter 9 (*Remote Procedure Calls*) there are several cases where a C++ *FunctionObject* is expected to execute on a destination process. In these cases the function arguments are serialized as described in this chapter. The FunctionObject itself (i.e. the `func` argument to `rpc`, `rpc_ff`, or `as_rpc`) is serialized in a type-dependent manner:

2 - A FunctionObject of a type defining non-trivial Serialization using the Class Serialization Interface (§6.2) is serialized as defined by that interface.

3 - A FunctionObject comprised of a simple function pointer or function reference is serialized via conversion to a function pointer offset into the source program's appropriate *code segment*[2].

4 - A FunctionObject comprised of a lambda expression is treated as TriviallySerializable, regardless of any actual contents of the object. As such, objects captured by copy in a lambda expression are transferred to the destination by making a byte copy. The behavior is undefined if the type of an object captured by copy is not TriviallySerializable.

5 The deserialized FunctionObject `dfunc` will be invoked by the library in a manner equivalent to `std::move(dfunc)(...)` (the details of the argument passing and type compatibility are specified in §9). In particular, FunctionObjects of class type must provide a type-compatible and publicly accessible overload of the function call operator `operator()`, whose member function qualifiers must not specify an lvalue ref-qualifier in the absence of `const`[3].

6 Further details of the implementation are not described here but typical allowed FunctionObjects include:

7 - C functions

8 - C++ global and file-scope functions

9 - Class static functions

10 - lambda expressions

---

[2]Transmission of raw function addresses across address spaces using trivial serialization would yield undefined behavior at the invocation point.

[3]For example, permitted combinations of member function qualifiers include omitted, `&&`, or `const &`.

6.6   Special Handling in Remote Procedure Calls

1   Remote procedure calls, whether standalone (§9) or completion-based (§7), perform special handling on certain non-Serializable `UPC++` data structures. Arguments that are either a reference to `dist_object` type (see §14 Distributed Objects) or a `team` (see §11 Teams) are transferred by their `dist_id` or `team_id` respectively. Execution of the RPC is deferred until all of the id's have a corresponding instance constructed on the recipient. When that occurs, `func` is enlisted for execution during user-level progress of the recipient's master persona (see §10 Progress), and it will be called with the recipient's instance references in place of those supplied at the send site. The behavior is undefined if the recipient's instance of a `dist_object` or `team` argument is destroyed before the RPC executes.

6.7   View-Based Serialization

1   `UPC++` also provides a mechanism for serializing the elements of a sequence, without explicitly serializing an enclosing container type. The following is an example of transferring a sequence with `rpc`:

```
1 std::list<double> items = /* fill with elements */;
2 auto fut = rpc_ff(1, [](view<double> packedlist) {
3   // target side gets object containing iterators
4   for (double elem : packedlist) {  // traverse network buffer
5     process(elem); // process each element
6   }
7 }, make_view(items.begin(), items.end()));
```

2   In this example, a `std::list<double>` contains the elements to be transferred. Calling `make_view` on its begin and end iterators results in a `view`, which can then be passed to a remote procedure call. The elements in the sequence are serialized and transferred as part of the RPC, and the target receives a `view` over the elements stored in the network buffer. The RPC can then iterate over the `view` to obtain each element.

3   There is an asymmetry in the `view` types at the initiator and target of an RPC, reflecting the difference in how the underlying sequences are stored in memory. In the example above, the type of the value returned by `make_view` is `view<double, std::list<double>::iterator>`, since the initiator supplies iterators associated with a list. The target of the RPC, however, receives a `view<double, view_default_iterator_t<double>>`, with the `view_default_iterator_t<T>` type representing an iterator over a network buffer.

The latter is the default argument for the second template parameter of `view`, so that a user can specify `view<T>` rather than `view<T, view_default_iterator_t<T>>`.

4 `UPC++` provides different handling of `view<T>` based on whether the element type `T` is TriviallySerializable or not. For TriviallySerializable element type, deserialization is a no-op, and the `view<T>` on the recipient is a direct view over a network buffer, providing both random access and access to the buffer itself. The corresponding `view_default_iterator_t<T>` is an alias for `T*`. On the other hand, if the `view` element type is not TriviallySerializable, then an element must be nontrivially deserialized before it can be accessed by the user. In such a case, the `view<T>` only provides access through an InputIterator, which deserializes and returns elements by value, and `view_default_iterator_t<T>` is an alias for `deserializing_iterator<T>`.

5 A `deserializing_iterator<T>` also provides several member functions that deserialize an object into existing memory. These avoid constructing a `deserialized_type_t<T>` on the stack and returning it by value, which can be inefficient for large types or types that are costly to move:

6 • `deserialize_overwrite(object)` deserializes an object into space occupied by an existing object of type `deserialized_type_t<T>`. The existing object is destructed before being overwritten by the deserialized object.

7 • `deserialize_into(optional_obj)` deserializes an object into a `upcxx::optional<deserialized_type_t<T>>`. The `upcxx::optional` is reset before a new object is read into the optional.

8 • `deserialize_into(pointer)` deserializes an object directly into the memory pointed to by `pointer`. Any existing object in the memory denoted by the pointer is not destructed. If `pointer` points to an existing object, `deserialize_overwrite(*pointer)` should be used instead.

9 The following is an example of using a `view` and `deserialize_into` to transfer a single object of a large, non-TriviallySerializable type:

```
1  BigObject *ptr = /* ... */;
2  future<> = rpc( target_rank ,
3    [](view<BigObject> item) {
4      auto opt = new upcxx::optional<BigObject>;
5      BigObject *ptr = item.begin().deserialize_into(opt);
6      /* ... consume the object */
7      delete opt;
8    },
9    make_view(ptr, ptr+1));
```

10 The result of deserializing a `view<T, Iter>` is always `view<T>`, even if `deserialized_type_t<T>` is some type `U` that is distinct from `T`[4]. In such a case, `T` is necessarily not TriviallySerializable, and `deserializing_iterator<T>` invokes the `deserialize` routine for `T` to produce an element of type `U`. The type `deserializing_iterator<T>::value_type` is an alias for the element type produced by `deserializing_iterator<T>`, and it is equivalent to `deserialized_type_t<T>`.

11 As a non-owning interface, `view` only provides `const` access to the elements in the underlying sequence, analogous to C++17 `string_view`. However, in the case of a `view<T>` that is received by the target of an RPC, where `T` is TriviallySerializable, the underlying elements are stored directly in a network buffer as indicated above. There is no external owning container, so `UPC++` permits a user to perform a `const_cast` conversion on an element and modify it.

12 The lifetime of the underlying data buffer and all view iterators on the target in both the TriviallySerializable and non-TriviallySerializable cases is restricted by default to the duration of the RPC. In this case, the elements must be processed or copied elsewhere before the RPC returns. However, if the RPC returns a future, then the lifetime of the buffer and view iterators is extended until that future is readied. This allows an RPC to initiate an asynchronous operation to consume the elements, and as long as the resulting future is returned from the RPC, the underlying buffer will remain valid until the asynchronous operation is complete and the future readied. Lifetime extension applies to all RPC variants, including `rpc_ff` and `as_rpc` where the return value is not made available to user code.

13 While `UPC++` manages the lifetime of the data underlying a `view` when it is an argument to an RPC, the library does not support a `view` as the return type of an RPC due to the lifetime issues it raises. Thus, an RPC is prohibited from returning a `view` even though it is classified as Serializable.

14 If a `view<T, IterType>` is passed to `rpc`, `rpc_ff`, or `as_rpc` where the type `T` is itself a `view`, behavior is undefined.

---

[4] Note this differs from the behavior of standard-library containers, where for example, `deserialized_type_t<std::vector<T>>` is `std::vector<deserialized_type_t<T>>`, and objects of the latter type are completely deserialized before being passed to RPC callbacks or completion events. The difference arises because the `view<T>` object passed to an RPC callback is a non-owning container over the serialized representation, and its `deserializing_iterator` performs deserialization on-demand using the deserialization code provided by `T`.

6.8 API Reference

```
1  template < typename T >
   struct is_trivially_serializable ;
```

2   Provides a member constant `value` that is `true` if T is TriviallySerializable and `false` otherwise. This trait may be specialized for user types.

```
3  template < typename T >
   struct is_serializable ;
```

4   Provides a member constant `value` that is `true` if T is Serializable and `false` otherwise. This trait may not be specialized. However, its value may be indirectly influenced by specializing `is_trivially_serializable<T>`, or implementing a class serialization interface for T (§6.2), as appropriate.

6.8.1 Optionals

```
1  template < typename T >
   class optional ;
   struct in_place_t ;
   constexpr in_place_t in_place ;
   struct nullopt_t ;
   constexpr nullopt_t nullopt ;
   class bad_optional_access ;
```

2   A class template that provides an implementation-defined subset of C++17 `std::optional` and related utilities [4]. If UPC++ is compiled with C++17 or later, these are aliases for the respective entities in the `std` namespace.

6.8.2   Views

```
1  template<typename T>
   class deserializing_iterator {
   public:
     // types
     using iterator_category = std::input_iterator_tag;
     using value_type        = deserialized_type_t<T>;
     using difference_type   = ptrdiff_t;
     using pointer           = value_type*;
     using reference         = value_type;

     deserializing_iterator();

     value_type operator*() const;
     pointer_type deserialize_overwrite(value_type& object) const;
     pointer_type deserialize_into(
                          optional<value_type>& optional_obj) const;
     pointer_type deserialize_into(void* pointer) const;

     deserializing_iterator& operator++();
     deserializing_iterator  operator++(int);
   };



2  // comparisons
   template<typename T>
   bool operator==(const deserializing_iterator& x,
                   const deserializing_iterator& y);
   template<typename T>
   bool operator!=(const deserializing_iterator& x,
                   const deserializing_iterator& y);
```

3   *C++ Concepts:* InputIterator

4   `T` must be Serializable.

5   An iterator over elements stored in a network buffer.   Dereferenc-
     ing the iterator causes the element to be deserialized and returned
     by value (i.e.   `deserializing_iterator<T>::reference` is an alias for
     `deserializing_iterator<T>::value_type`).

⁶ While this iterator is classified as an InputIterator, it does not support `operator->`, as the underlying element must be materialized on demand and its lifetime would not extend beyond the application of the operator.

⁷
```cpp
template<typename T>
deserialized_type_t<T>*
  deserializing_iterator<T>::deserialize_overwrite(
                            deserialized_type_t<T>& object) const;
```

⁸ *Precondition:* This iterator must be pointing to a valid element.

⁹ Destructs `object`, reads the serialized representation of an object of type `T` referenced by this iterator, and deserializes it into the memory denoted by `&object`. Returns a pointer to the newly constructed object.

¹⁰
```cpp
template<typename T>
deserialized_type_t<T>*
  deserializing_iterator<T>::deserialize_into(
        optional<deserialized_type_t<T>>& optional_obj) const;
```

¹¹ *Precondition:* This iterator must be pointing to a valid element. `deserialized_type_t<T>` must not have a top-level `const` qualifier. If `T` is TriviallySerializable, then `T` must also be DefaultConstructible. If custom serialization is defined for `T`, it must not use the deprecated interface for the `deserialize` member-function template.

¹² Reads the serialized representation of an object of type `T` referenced by this iterator and deserializes it into `optional_obj`. `optional_obj` is reset before the deserialized object is constructed into the optional. Returns a pointer to the newly constructed object.

¹³
```cpp
template<typename T>
deserialized_type_t<T>*
  deserializing_iterator<T>::deserialize_into(void* pointer) const;
```

¹⁴ *Precondition:* This iterator must be pointing to a valid element. `pointer` must point to a location with appropriate size and alignment for an object of type `deserialized_type_t<T>`.

¹⁵ Reads the serialized representation of an object of type `T` referenced by this iterator and deserializes it into the memory denoted by `pointer`. Returns a pointer to the newly constructed object.

```
16  template<typename T>
    using view_default_iterator_t = /* ... */;
```

17   A type alias that is equivalent to `T*` if `T` is TriviallySerializable (i.e. `upcxx::is_trivially_serializable<T>::value` is true), and `deserializing_iterator<T>` otherwise.

```
18  template<typename T, typename IterType=view_default_iterator_t<T>>
    class view {
    public:
      // types
      using iterator  = IterType;
      using size_type = size_t;

      // iterators
      iterator begin();
      iterator end();

      // capacity
      size_type size() const;
    };
```

19   *C++ Concepts:* DefaultConstructible, CopyConstructible, CopyAssignable, Destructible

20   `UPC++` *Concepts:* Serializable

21   A class template representing a view over an underlying sequence of elements of type `T`, delimited by `begin()` and `end()`.

```
22  template<typename T>
    class view<T, T*> {
    public:
      // types
      using value_type            = T;
      using pointer               = T*;
      using const_pointer         = const T*;
      using reference             = T&;
      using const_reference       = const T&;
      using const_iterator        = const T*;
      using iterator              = const_iterator;
      using const_reverse_iterator =
            std::reverse_iterator<const_iterator>;
      using reverse_iterator      = const_reverse_iterator;
```

```
using size_type           = size_t;
using difference_type     = ptrdiff_t;

// no explicit construct/copy/destroy for non-owning type

// iterators
const_iterator         begin() const;
const_iterator         cbegin() const;
const_iterator         end() const;
const_iterator         cend() const;
const_reverse_iterator rbegin() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator rend() const;
const_reverse_iterator crend() const;

// capacity
bool empty() const;
size_type size() const;

// element access
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
const_reference front() const;
const_reference back() const;

const_pointer data() const;
};
```

23   *C++ Concepts:* DefaultConstructible, CopyConstructible, CopyAssignable, Destructible

24   `UPC++` *Concepts:* Serializable

25   A template specialization representing a view over a network buffer of elements of TriviallySerializable type `T`, delimited by `begin()` and `end()`.

26   *Exceptions:* `at(n)` throws `std::out_of_range` if `n` is not in the range `[0, size())`.

27
```
template<typename T, typename IterType>
view<T, IterType>::view();
```

28   *Precondition:* `IterType` must satisfy the ForwardIterator C++ concept. The type `std::iterator_traits<IterType>::value_type` must be the same as `T`. `T` must be Serializable.

29   Initializes this `view` to represent an empty sequence.

```
30  template < typename IterType >
    view < typename std :: iterator_traits < IterType >:: value_type , IterType >
      make_view ( IterType begin , IterType end );
```

31   *Precondition:* `IterType` must satisfy the ForwardIterator C++ concept. The underlying element type (`std::iterator_traits<IterType>::value_type`) must be Serializable.

32   Constructs a `view` over the sequence delimited by `begin` and `end`.

```
33  template < typename IterType >
    view < typename std :: iterator_traits < IterType >:: value_type , IterType >
      make_view ( IterType begin , IterType end , size_t num_items );
```

34   *Precondition:* `IterType` must satisfy the ForwardIterator C++ concept. The underlying element type (`std::iterator_traits<IterType>::value_type`) must be Serializable. `num_items` must be equal to the number of elements in `[begin, end)`.

35   Constructs a `view` over the sequence delimited by `begin` and `end`. This is semantically equivalent to `make_view(begin, end)`, but it may provide better performance when `IterType` is not a RandomAccessIterator.

```
36  template < typename Container >
    view < typename Container :: value_type ,
        typename Container :: const_iterator >
      make_view ( const Container & container );
```

37   *Precondition:* `Container` must satisfy the Container C++ concept. The underlying element type (`Container::value_type`) must be Serializable.

38   Constructs a `view` over the sequence delimited by `container.cbegin()` and `container.cend()`.

6.8.3 Class Serialization

```
1  template < typename T >
   struct serialization_traits ;
```

2   *Precondition:* T must be Serializable or a (possibly multidimensional) array type whose element type is Serializable.

3   Provides a member type alias `deserialized_type` that is the type resulting from deserializing an object that was serialized as the type `T`. If `T` is an array type `U[n]`, then `deserialized_type` is an alias for `serialization_traits<U>` `::deserialized_type[n]`.

```
4  template < typename T >
   using deserialized_type_t =
       typename serialization_traits <T >:: deserialized_type ;
```

5   Type alias for `serialization_traits<T>::deserialized_type`.

```
6  template < typename T >
   static deserialized_type_t <T >
       serialization_traits <T >:: deserialized_value ( T const & object );
```

7   *Precondition:* T must be Serializable and must not be a `view`. `deserialized_type_t<T>` must be Movable.

8   Returns a value that is the deserialized counterpart of `object`. Equivalent to serializing `object` into temporary storage and deserializing the result.

```
9  # define UPCXX_SERIALIZED_FIELDS (...) /* see below */
```

10   A variadic macro for specifying a subset of member variables to be automatically serialized by `UPC++`, as described in §6.2.1.

```
11 # define UPCXX_SERIALIZED_VALUES (...) /* see below */
```

12   A variadic macro for specifying a set of values to be used by `UPC++` to serialize an object, as described in §6.2.2.

```
13 # define UPCXX_SERIALIZED_DELETE () /* see below */
```

14   A macro for disabling serialization of a derived class when the derived class would otherwise inherit a serialization mechanism from a base class (§6.2.5).

```
15 #define UPCXX_SERIALIZED_BASE(base) /* see below */
```

16 A macro for specifying serialization of a base subobject of type `base`, as described in §6.2.5.

```
17 template<typename T>
   struct serialization;
```

18 A class template that can be specialized to customize serialization and deserialization for a type `T`, as described in §6.2.3.

```
19 template<typename T>
   void [Writer]::write(T const &object);
```

20 *Precondition:* `T` must be Serializable or a (possibly multidimensional) array type whose element type is Serializable.

21 Writes a serialized representation of `object` to the given writer.

22 *Advice to users:* An invocation of `write<U[n]>` is not guaranteed to be equivalent to a sequence of invocations of `write<U>`. Thus, the only specified way to read an object written by `write<U[n]>` is via a call to `read_overwrite<U[n]>` or `read_into<U[n]>`.

```
23 template<typename IterType>
   size_t [Writer]::write_sequence(IterType begin, IterType end);
```

24 *Precondition:* `IterType` must satisfy the ForwardIterator C++ concept. The underlying element type (`std::iterator_traits<IterType>::value_type`) must be Serializable.

25 Writes serialized representations of the objects in the sequence delimited by `begin` and `end` to the given writer and returns the number of objects written. This is semantically equivalent to separate calls to `write` on the elements of the sequence in order, but it may provide better performance.

```
26  template < typename IterType >
    size_t [Writer]::write_sequence(IterType begin, IterType end,
                                         size_t num_items);
```

27   *Precondition:* `IterType` must satisfy the ForwardIterator C++ concept. The underlying element type (`std::iterator_traits<IterType>::value_type`) must be Serializable. `num_items` must be equal to the number of elements in `[begin, end)`.

28   Writes serialized representations of the objects in the sequence delimited by `begin` and `end` to the given writer and returns the number of objects written. This is semantically equivalent to `write_sequence(begin, end)`, but it may provide better performance when `IterType` is not a RandomAccessIterator.

```
29  template < typename T >
    struct [Writer]::reserve_handle;
```

30   *C++ Concepts:* Movable, Destructible

31   Represents a reserved location in a writer where an object of type `T` is to be written.

```
32  template < typename T >
    typename [Writer]::reserve_handle<T> [Writer]::reserve();
```

33   *Precondition:* `T` must be TriviallySerializable

34   Reserves space in the given writer for an object of type `T` and returns a handle to the resulting location. The handle must be written by a call to `commit` prior to the return of the function that invoked `reserve` to create the handle.

```
35  template < typename T >
    void [Writer]::commit(
                    typename [Writer]::reserve_handle<T>&& handle,
                    T const& object);
```

36   *Precondition:* `T` must be TriviallySerializable. `handle` must have been constructed through a call to `reserve` on this writer, and it must not previously have had `commit` called on it.

37   Writes a representation of `object` to the location in the writer denoted by `handle`. Invalidates `handle`.

```
38  template < typename T>
    deserialized_type_t <T> [Reader]::read();
```

39   *Precondition:* The current position of the reader must be a location where an object of type `T` was written. `T` must not be an array type.

40   Reads a serialized representation of an object of type `T` and returns a deserialized object of type `deserialized_type_t<T>`.

```
41  template < typename T>
    deserialized_type_t <T>* [Reader]::read_overwrite(
                                    deserialized_type <T>& object);
```

42   *Precondition:* The current position of the reader must be a location where an object of type `T` was written. `deserialized_type_t<T>` must not have a top-level `const` qualifier.

43   Destructs `object`, reads a serialized representation of an object of type `T`, and constructs a deserialized object of type `deserialized_type_t<T>` in the memory denoted by `&object`. Returns a pointer to the newly constructed object.

```
44  template < typename T>
    deserialized_type_t <T>* [Reader]::read_into(void* pointer);
    template < typename T>
    deserialized_type_t <T>* [Reader]::read_into(
                      optional < deserialized_type_t <T>>& optional_obj);
```

Preconditions:

45   • The current position of the reader must be a location where an object of type `T` was written.

46   • In the first variant, `pointer` must point to a location with appropriate space and alignment for an object of type `deserialized_type_t<T>`. If `deserialized_type_t<T>` is not TriviallyDestructible, then an existing object must not be located at the memory denoted by `pointer`.

47   • In the second variant, `deserialized_type_t<T>` must not have a top-level `const` qualifier. If `T` is TriviallySerializable, then `T` must also be DefaultConstructible. If custom serialization is defined for `T`, it must not use the deprecated interface for the `deserialize` member-function template.

48   Reads a serialized representation of an object of type `T` and constructs a deserialized object of type `deserialized_type_t<T>` in the location denoted by `pointer` or `optional_obj`. Returns a pointer to the newly constructed object.

49   In the second variant, `optional_obj` is reset before the deserialized object is constructed into the optional.

50
```
template<typename T>
deserialized_type_t<T>* [Reader]::read_sequence_overwrite(
                    deserialized_type_t<T>* array, size_t num_items);
```

51   If `num_items` is zero, `array` is returned (and otherwise ignored), and the call has no preconditions or effects.

52   *Precondition:* The current position of the reader must be a location where `num_items` objects of type `T` were written. `array` must point to a location with existing objects of type `deserialized_type_t<T>`.

53   Destructs the existing `num_items` objects located at the memory denoted by `array`, reads serialized representations of `num_items` objects of type `T`, and constructs an array of deserialized objects of type `deserialized_type_t<T>` in the memory denoted by `array`. Returns a pointer to the first newly constructed object in the array.

54
```
template<typename T>
deserialized_type_t<T>*
    [Reader]::read_sequence_into(void* pointer, size_t num_items);
```

55   If `num_items` is zero, `pointer` is returned after being converted to a `deserialized_type_t<T>*` via a `reinterpret_cast` (and `pointer` is otherwise ignored), and the call has no preconditions or effects.

56   *Precondition:* The current position of the reader must be a location where `num_items` objects of type `T` were written. `pointer` must point to a location with appropriate space and alignment for an array of `num_items` objects of type `deserialized_type_t<T>`. If `deserialized_type_t<T>` is not TriviallyDestructible, then existing objects must not be located at the memory denoted by `pointer`.

57   Reads serialized representations of `num_items` objects of type `T` and constructs an array of deserialized objects of type `deserialized_type_t<T>` in the memory denoted by `pointer`. Returns a pointer to the first newly constructed object in the array.

```
58  template <typename ...Args >
    /* see below */ [Storage]::construct(Args &&...args);
```

59   *Precondition:* The constructor selected by the invocation `U(std::forward<Args>(args)...)` must have public member access.

60   Forwards `args` to the constructor for the underlying object type `U` and returns a pointer to the newly constructed object. The type `U` is `deserialized_type<T>` for a Storage object that is passed to the `deserialize` member-function template for type `T`.

61   *Advice to users:* The *passkey idiom* can be used to prevent external access to a public constructor. The following is an example of this pattern:

```
1  class UserType {
2    struct Key {                              // private member type
3      // explicit to prevent conversion from initializer list
4      explicit Key() {}
5    };
6  public:
7    UserType(Key key /*, other parameters */);
8    struct upcxx_serialization {
9      /* ... */
10     template <typename Reader, typename Storage >
11     static UserType* deserialize(Reader& reader,
12                                  Storage storage) {
13       return storage.construct(Key{} /*, other arguments */);
14     }
15   };
16 };
```

Only code within `UserType` can construct a `Key` object, so outside code cannot invoke the constructor that takes a `Key` without having an existing `Key` object available. The `deserialize` member-function template makes such an object available to `construct`, allowing it to invoke the constructor.

Chapter 7

Completion

7.1 Overview

1 Asynchronous operations such as communication entail the concept of completion, meaning that the effect of the operation is now visible on the source or target process and that resources, such as memory on the source and destination sides, are no longer in use by `UPC++`. A single `UPC++` call may have several completion events associated with it, indicating completion of different stages of a communication operation. These events are categorized as follows:

2 • *Source completion:* The source-side resources of a communication operation are no longer in use by `UPC++`, and the application is now permitted to modify or reclaim them.

3 • *Remote completion:* The data have been deposited on the remote target process, and they can be consumed by the target.

4 • *Operation completion:* The operation is complete from the viewpoint of the initiator. The transferred data can now be read by the initiator, resulting in the values that were written to the target locations.

5 A completion event may be associated with some values produced by the communication operation, or it may merely signal completion of an action. Each communication operation specifies the set of completion events it provides, as well as the values that a completion event produces. Unless otherwise indicated, a completion event does not produce a value.

6 `UPC++` provides several alternatives for how completion can be signaled to the program:

7   • *Future:* The communication call returns a future, which will be readied after the completion event occurs. This is the default notification mode for communication operations. If the completion event is associated with some values of type `T...`, then the returned future will have type `future<T...>`. If no value is associated with the completion, then the future will have type `future<>`.

8   • *Promise:* The user provides a promise when requesting notification of a completion event. Initiating a communication operation increments the dependency count of the promise, and that promise will have one its dependencies fulfilled after the event occurs. The promise must have a non-zero dependency count. If the completion event is associated with some values of type `T...`, then it must be valid to call `fulfill_result()` on the promise with values of type `T...`, and the promise must not have had `fulfill_result()` called on it. The promise will then have `fulfill_result()` called on it with the associated values after the completion event occurs. If no value is associated with the completion, then the promise may have any type. It will have an anonymous dependency fulfilled upon the completion event.

9   • *Local-Procedure Call (LPC):* The user provides a target persona and a callback function object when requesting notification of a completion event. If the completion is associated with some values of type `T...`, then the callback must be invokable with a sequence of `sizeof...(T)` arguments, where if the nth component of `T` has type `U`, then the nth argument provided to the callback has type:

10   – `U` if `U` is of reference type

11   – `U&&` if `U` is of non-reference type

12   If the completion is not associated with any values, the callback must be invokable with no arguments. The callback, together with the associated completion values if any, is enlisted for execution during user-level progress of the given persona after the completion event occurs.

13   • *Remote-Procedure Call (RPC):* The user provides a function object that is either Serializable or one of the allowed FunctionObjects in §6.5 when requesting notification of a completion event, as well as the arguments on which the function object should be invoked. Each argument must either be Serializable, a *cq* `dist_object<T>&`, or *cq* `team&`. The result of serializing and deserializing the function object must be invokable on the values that result from serializing and deserializing the arguments, and the invocation must not throw an exception. Specifically, if an argument is of type *cq* `dist_object<T>&` or *cq* `team&`, the function object must accept a value of `dist_object<T>&` or `team&`, respectively, for the associated parameter. If the argument is of some other type `T`, the function object must accept a value of type `deserialized_type_t<T>&&` for the respective parameter. The function object and

arguments are transferred as part of the communication operation, and the invocation is enlisted for execution during user-level progress of the master persona of the target process after the completion event occurs.

14 The result from invoking the function object is discarded. However, the return value affects the lifetime of the deserialized objects that the UPC++ runtime constructs[1] and passes to the function object. If the return value is a non-future value or a ready future, the deserialized objects are destructed immediately after the invocation returns. On the other hand, if the return value is a non-ready future, destruction of the deserialized objects is deferred until after the future becomes ready, allowing the function object to safely initiate further asynchronous computation that operates on those objects.

15 - *Buffered:* The communication call consumes the source-side resources of the operation before the call returns, allowing the application to immediately modify or reclaim them. This delays the return of the call until after the source-completion event. The implementation may internally buffer the source-side resources or block until network resources are available to inject the data directly.

16 - *Blocking:* This is similar to buffered completion, except that the implementation is required to block until network resources are available to inject the data directly.

17 Future, promise, and LPC completions are only valid for completion events that occur at the initiator of a communication call, namely source and operation completion. RPC completion is only valid for a completion event that occurs at the target of a communication operation, namely remote completion. Buffered and blocking completion are only valid for source completion. More details on futures and promises are in Ch. 5, while LPC and RPC callbacks are discussed in Ch. 10.

18 If a completion event is associated with some values, the UPC++ runtime is permitted to pass separate copies of those values to each notification registered on that completion event. If multiple notifications are registered on an event that produces values, their respective types must be CopyConstructible. Otherwise, the respective types must be MoveConstructible.

19 Notification of future and promise completions may be *deferred* or *eager* [7]:

20 - *Deferred:* Notification only happens during user-level progress of the initiator, even if a completion event occurs synchronously (i.e. prior to the return from initiation). Only when the initiating thread (persona actually) enters user-level progress will the future or promise be eligible for taking on a readied or fulfilled state.

---

[1]This excludes `team&` and `dist_object<T>&` arguments, where the underlying objects are not constructed by the UPC++ runtime in deserialization.

21 • *Eager:* If the completion event occurs synchronously, the `UPC++` implementation is permitted (but not required) to perform notification of completion immediately. If completion is not signaled immediately, then it must be signaled at some later point during user-level progress of the initiator.

22 Notification of other kinds of completion are always deferred until user-level progress of the target process or persona. See Ch. 10 for the full discussion on user progress and personas.

23 If buffered or blocking completion is requested, then the source-completion event occurs before the communication call returns. However, deferred source-completion notifications, such as readying a deferred future completion or executing an LPC, are still delayed until the next user-level progress. Similarly, deferred source-completion notifications for empty data transfers (e.g. `rput` with a size of zero) are delayed until the next user-level progress.

24 Operation completion implies both source and remote completion. However, it does not imply that notifications associated with source and remote completion have occurred. Similarly, remote completion implies source completion, but it does not imply that notifications associated with source completion have occurred.

7.2   Completion Objects

1 The `UPC++` mechanism for requesting notification of completion is through opaque *completion objects*, which associate notification actions with completion events. Completion objects are CopyConstructible, CopyAssignable, and Destructible, and the same completion object may be passed to multiple communication calls. A simple completion object is constructed by a call to a static member function of the `source_cx`, `remote_cx`, or `operation_cx` class, providing notification for the corresponding event. Member functions such as `as_future`, `as_promise`, `as_lpc`, and `as_rpc` request notification through a future, promise, LPC, or RPC, respectively. Only the member functions that correspond to valid means of signaling notification of an event are defined in the class associated with that event.

2 The following is an example of a simple completion object:

```
1 global_ptr<int> gp1 = /* some global pointer */;
2 promise<int> pro1;
3 auto cxs = operation_cx::as_promise(pro1);
4 rget(gp1, cxs);
5 pro1.finalize(); // fulfill the initial anonymous dependency
```

3 The `rget` function, when provided just a `global_ptr<int>`, transfers a single `int` from the given location to the initiator. Thus, operation completion is associated with an `int` value, and the promise used for signaling that event must have type compatible with an `int` value, e.g. `promise<int>`. The user constructs a completion object that requests operation notification on the promise `pro1` by calling `operation_cx::as_promise(pro1)`. Since a completion object is opaque, the `auto` keyword is used to deduce the type of the completion object. The resulting completion object can then be passed to `rget`, which fulfills the promise with the transferred value upon operation completion.

4 A user can request notification of multiple completion events, as well as multiple notifications of a single completion event. The pipe (`|`) operator can be used to combine completion objects to construct a union of the operands. The following is an example:

```
1  int foo() {
2    return 0;
3  }
4
5  int bar(int x) {
6    return x;
7  }
8
9  void do_comm(double *src, size_t count) {
10   global_ptr<double> dest = /* some global pointer */;
11   promise<> pro1;
12   persona &per1 = /* some persona */;
13   auto cxs = (operation_cx::as_promise(pro1) |
14               source_cx::as_future() |
15               operation_cx::as_future() |
16               operation_cx::as_future() |
17               source_cx::as_lpc(per1, foo) |
18               remote_cx::as_rpc(bar, 3)
19               );
20   std::tuple<future<>, future<>, future<>> result =
21     rput(src, dest, count, cxs);
22   pro1.finalize().wait(); // finalize promise, wait on its future
23  }
```

5 This code initiates an `rput` operation, which provides source-, remote-, and operation-completion events. A unified completion object is constructed by applying the pipe operator to individual completion objects. When `rput` is invoked with the resulting unified completion object, it returns a tuple of futures corresponding to the individual future completions requested. The ordering of futures in this tuple matches the order of application

of the pipe operator (this operator is associative but not commutative). In the example above, the first future in the tuple would correspond to source completion, and the second and third would be for operation completion. If no future-based notification is requested, then the return type of the communication call would be `void` rather than a tuple.

6   When multiple notifications are requested for a single event, the order in which those notifications occur is unspecified. In the code above, the order in which `pro1` is fulfilled and the two futures for operation completion are readied is indeterminate. More generally, whenever two or more event notifications are pending (regardless of what events they correspond to), the order in which those notifications are delivered is unspecified. In the code above, for example, operation-completion notifications might be delivered before source-completion notifications, despite the fact that operation completion implies source completion.

7   Unlike a direct call to the `rpc` function (Ch. 9), but like a call to `rpc_ff`, an RPC completion callback does not return a result to the initiator. Thus, the value returned by the RPC invocation of `bar` above is discarded.

8   Arguments to `remote_cx::as_rpc` are serialized at an unspecified time between the invocation of `as_rpc` and the return from the invocation of a communication operation that accepts the resulting completion object. If multiple communication operations use a single completion object resulting from `as_rpc`, then the arguments may be serialized multiple times. For lvalue arguments, the user must ensure that they remain valid until the return from the invocation of all communication operations that use the associated completion object. Rvalue arguments are guaranteed to be "consumed" before the `remote_cx::as_rpc` factory function returns the completion object, meaning they are move-constructed into an internal location and/or fully serialized.

7.2.1   Restrictions

1   The API reference for a `UPC++` call that supports the completion interface lists the completion events that the call provides, as well as the types of values associated with each event, if any. The result is undefined if a completion object is passed to a call and the object contains a request for an event that the call does not support. Passing a completion object that contains a request whose type does not match the types provided by the corresponding completion event, as described in §7.1, also results in undefined behavior.

2   If a `UPC++` call provides both operation and remote completion, then at least one must be requested by the provided completion object. If a call provides operation but not remote completion, then operation completion must be requested. The behavior of the program is

undefined if neither operation nor remote completion is requested from a call that supports one or both of operation or remote completion.

3 A promise object associated with a promise-based completion request must have a dependency count greater than zero when the completion object is passed to a `UPC++` operation. The result is undefined if the same promise object is used in multiple requests for notifications that produce values.

7.2.2   Completion and Return Types

1 In subsequent API-reference sections, the opaque type of a completion object is denoted by `CType`. Similarly, `RType` denotes a return type that is dependent on the completion object passed to a `UPC++` call. This return type is as follows:

2 • `void`, if no future-based completions are requested

3 • `future<T...>`, if a single future-based completion is requested, where `T...` is the sequence of types associated with the given completion event

4 • `std::tuple<future<T...>...>`, if multiple future-based completions are requested, where each future's arguments `T...` is the sequence of types associated with the corresponding completion event

5 Type deduction, such as with `auto`, is recommended when working with completion objects and return types.

7.2.3   Default Completions

1 If a completion object is not explicitly provided to a communication call, then a default completion object is used. For most calls, the default is `operation_cx::as_future()`. However, for `rpc_ff`, the default completion is `source_cx::as_buffered()`, and for `rpc`, it is `source_cx::as_buffered() | operation_cx::as_future()`. The default completion of a `UPC++` communication call is listed in its API reference as a defaulted function argument, for example:

```
Cx &&completions=operation_cx::as_future()
```

indicates a default completion of `operation_cx::as_future()`. Each such function template also includes a template argument corresponding to the type of this completion argument; this template argument is usually inferred, and the default type is omitted for brevity of presentation.

7.3 API Reference

1
```
struct source_cx;

struct remote_cx;

struct operation_cx;
```

2   Types that contain static member functions for constructing completion objects for source, remote, and operation completion.

3
```
static CType source_cx::as_future();

static CType operation_cx::as_future();
```

4   Constructs a completion object that represents notification of source or operation completion with a future. It is implementation-defined whether the completion objects created by these functions will request eager or deferred notification.

5
```
static CType source_cx::as_defer_future();

static CType operation_cx::as_defer_future();
```

6   Constructs a completion object that represents deferred notification of source or operation completion with a future.

7
```
static CType source_cx::as_eager_future();

static CType operation_cx::as_eager_future();
```

8   Constructs a completion object that represents eager notification of source or operation completion with a future.

```
9  template < typename ...T >
   static CType source_cx :: as_promise ( promise <T...> pro );

   template < typename ...T >
   static CType operation_cx :: as_promise ( promise <T...> pro );
```

10   *Precondition:* `pro` must have a dependency count greater than zero.

11   Constructs a completion object that represents signaling the given promise
      upon source or operation completion. It is implementation-defined whether
      the completion objects created by these functions will request eager or deferred
      notification.

```
12  template < typename ...T >
    static CType source_cx :: as_defer_promise ( promise <T...> pro );

    template < typename ...T >
    static CType operation_cx :: as_defer_promise ( promise <T...> pro );
```

13   *Precondition:* `pro` must have a dependency count greater than zero.

14   Constructs a completion object that represents deferred notification of source
      or operation completion via signaling the given promise.

```
15  template < typename ...T >
    static CType source_cx :: as_eager_promise ( promise <T...> pro );

    template < typename ...T >
    static CType operation_cx :: as_eager_promise ( promise <T...> pro );
```

16   *Precondition:* `pro` must have a dependency count greater than zero.

17   Constructs a completion object that represents eager notification of source or
      operation completion via signaling the given promise.

```
18  template < typename Func >
    static CType source_cx :: as_lpc ( persona &target , Func &&func );

    template < typename Func >
    static CType operation_cx :: as_lpc ( persona &target , Func &&func );
```

19   *Precondition:* `Func` must be a function-object type, and the underlying decayed
      type (`std::decay<Func&&>::type`) must be CopyConstructible. `func` must
      not throw an exception when invoked.

20   Constructs a completion object that represents the enqueuing of `func` on the
      given local persona upon source or operation completion.

21
```
template < typename Func , typename ... Args >
static CType remote_cx :: as_rpc ( Func && func , Args ... && args );
```

      Preconditions:

22   • `Func` must be a function-object type.

23   • If `Func&&` is an rvalue-reference type, the underlying decayed type
        (`std::decay<Func&&>::type`) must be CopyConstructible.

24   • `Func` must either be Serializable or one of the FunctionObjects listed in
        §6.5.

25   • `deserialized_type_t<Func>` must be MoveConstructible.

26   • Each of `Args...` must either be a Serializable type, or *cq*
        `dist_object<T>&`, or *cq* `team&`.

27   • If a type `Arg` in `Args&&...` is an rvalue-reference type, the underlying
        decayed type (`std::decay<Arg>::type`) must be CopyConstructible.

28   • If a type `Arg` in `Args&&...` is not *cq* `dist_object<T>&` or *cq* `team&`,
        `deserialized_type_t<Arg>` must be MoveConstructible.

29   • The invocation of the deserialized function object on the deserialized ar-
        guments must not throw an exception.

30   Constructs a completion object that represents the enqueuing of `func` on a
      target process upon remote completion.

31
```
static CType source_cx :: as_buffered ();
```

32   Constructs a completion object that represents buffering source-side resources
      or blocking until they are consumed before a communication call returns, de-
      laying the return until the source-completion event occurs.

```
33 static CType source_cx::as_blocking();
```

34   Constructs a completion object that represents blocking until source-side resources are consumed before a communication call returns, delaying the return until the source-completion event occurs.

```
35 template<typename CTypeA, CTypeB>
   CType operator|(CTypeA &&a, CTypeB &&b);
```

36   *Precondition:* CTypeA and CTypeB must be completion types.

37   Constructs a completion object that is the union of the completions in `a` and `b`. Future-based completions in the result are ordered the same as in `a` and `b`, with those in `a` preceding those in `b`.

Chapter 8

One-Sided Communication

8.1 Overview

1 The main one-sided communication functions for `UPC++` are `rput` and `rget`. Where possible, the underlying transport layer will use RDMA techniques to provide the lowest-latency transport possible. The type `T` used by `rput` or `rget` needs to be **TriviallySerializable**, as described in Chapter 6 (*Serialization*).

8.2 API Reference

8.2.1 Remote Puts

1 ```
template<typename T, typename Cx=/*...*/>
RType rput(T value, global_ptr<T> dest,
           Cx &&completions=operation_cx::as_future());
```

2     *Precondition:* `T` must be TriviallySerializable.

3     Initiates a transfer of `value` that will store it in the memory referenced by `dest`.

4     Remote-completion operations execute on the master persona of the process associated with the destination (i.e. `dest.where()`).

*Completions:*

5 • *Remote*: Indicates completion of the transfer of `value`.

6 • *Operation*: Indicates completion of all aspects of the operation: the transfer and remote stores are complete.

7 *C++ memory ordering:* The writes to `dest` will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment) and remote-completion actions (RPC enlistment). For LPC and RPC completions, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of the completion function.

8 *UPC++ progress level:* `internal`

```
9  template<typename T, typename Cx=/*...*/)
   RType rput(T const *src, global_ptr<T> dest, size_t count,
             Cx &&completions=operation_cx::as_future());
```

10 *Precondition:* `T` must be TriviallySerializable. The source and destination memory regions must not overlap. `src` and `dest` must not be null pointers, even if `count` is zero.

11 Initiates an operation to transfer and store the `count` items of type `T` beginning at `src` to the memory beginning at `dest`. The values referenced in the `[src,src+count)` interval must not be modified until either source or operation completion is indicated.

12 Remote-completion operations execute on the master persona of the process associated with the destination (i.e. `dest.where()`).

Completions:

13 • *Source*: Indicates completion of injection or internal buffering of the source values, signifying that the `src` buffer may be modified.

14 • *Remote*: Indicates completion of the transfer of the values, implying readiness of the target buffer `[dest,dest+count)`.

15 • *Operation*: Indicates completion of all aspects of the operation: the transfer and remote stores are complete.

16   *C++ memory ordering:* The reads of `src` will have a *happens-before* relationship with the source-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment). The writes to `dest` will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment) and remote-completion actions (RPC enlistment). For LPC and RPC completions, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of the completion function.

17   *UPC++ progress level:* `internal`

8.2.2   Remote Gets

```
1  template<typename T, typename Cx=/*...*/>
   RType rget(global_ptr<const T> src,
              Cx &&completions=operation_cx::as_future());
```

2   *Precondition:* `T` must be TriviallySerializable.

3   Initiates a transfer to this process of a single value of type `T` located at `src`. The value will be transferred to the calling process and delivered in the operation-completion notification.

       Completions:

4   • *Operation*: Indicates completion of all aspects of the operation, including transfer and readiness of the resulting value. This completion produces a value of type `T`.

5   *C++ memory ordering:* The read of `src` will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment). All evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of any LPC associated with operation completion.

6   *UPC++ progress level:* `internal`

```
7  template<typename T, typename Cx=/*...*/>
   RType rget(global_ptr<const T> src, T *dest, size_t count,
              Cx &&completions=operation_cx::as_future());
```

8   *Precondition:* `T` must be TriviallySerializable. The source and destination memory regions must not overlap. `src` and `dest` must not be null pointers, even if `count` is zero.

9   Initiates a transfer of `count` values of type `T` beginning at `src` and stores them in the locations beginning at `dest`. The source values must not be modified until operation completion is notified.

     Completions:

10   • *Operation*: Indicates completion of all aspects of the operation, including transfer and readiness of the resulting values.

11   *C++ memory ordering:* The reads of `src` and writes to `dest` will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment). All evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of any LPC associated with operation completion.

12   *UPC++ progress level:* `internal`

Chapter 9

Remote Procedure Call

9.1   Overview

1   `UPC++` provides remote procedure calls (RPCs) for injecting function calls into other pro-
cesses. These injections are one-sided, meaning the recipient is not required to explicitly
acknowledge which functions are expected. Concurrent with a process's execution, incom-
ing RPCs accumulate in an internal queue managed by `UPC++`. The only control a process
has over inbound RPCs is when it would like to check its inbox for arrived function calls
and execute them. Draining the RPC inbox is one of the many responsibilities of the
progress API (see Ch. 10, *Progress*).

2   There are two main flavors of RPC in `UPC++`: *fire-and-forget* (`rpc_ff`) and *round trip* (`rpc`).
Each takes a function `Func` together with variadic arguments `Args`.

3   The `rpc_ff` call serializes the given function and arguments into a message destined for the
recipient, and guarantees that this function call will be placed eventually in the recipient's
inbox. The round-trip `rpc` call does the same, but also forces the recipient to reply to the
sender of the RPC with a message containing the return value of the function, providing
the value for operation completion of the sender's invocation of `rpc`. Thus, when the future
is ready, the sender knows the recipient has executed the function call. Additionally, if the
return value of `func` is a future, the recipient will wait for that future to become ready
before sending its result back to the sender.

4   There are important restrictions on what the permissible types for `func` and its bound
arguments can be for RPC functions. First, the `Func` type must be a Callable func-
tion object Second, `Func` must either be Serializable or one of the FunctionObject types
listed in §6.5, and all `Args...` types must be Serializable (see Ch. 6, *Serialization*), a

cq dist_object<T>&, or *cq* team&.   Third, the result of serializing and deserializing the function object (a value of type deserialized_type_t<Func>) must be invokable on the values that result from serializing and deserializing the arguments.   Specifically, if an argument is of type *cq* dist_object<T>& or *cq* team&, the function object must accept a value of dist_object<T>& or team&, respectively, for the associated parameter. If the argument is of some other type T, the function object must accept a value of type deserialized_type_t<T>&& for the respective parameter[1].   Lastly, the invocation must not throw an exception.

9.2   Remote Hello World Example

1   Figure 9.1 shows a simple alternative *Hello World* example where each process issues an rpc to its neighbor, where the last rank wraps around to 0.

```cpp
1 #include <upcxx/upcxx.hpp>
2 #include <iostream>
3 void hello_world(intrank_t num){
4   std::cout << "Rank " << num <<"  told rank " << upcxx::rank_me()
5       << " to say Hello World" << std::endl;
6 }
7 int main(int argc, char** argv[]){
8   upcxx::init();              // Start UPC++ state
9   intrank_t remote = (upcxx::rank_me()+1)%upcxx::rank_n();
10   auto f = upcxx::rpc(remote, hello_world, upcxx::rank_me());
11   f.wait();
12   upcxx::finalize();       // Close down UPC++ state
13   return 0;
14 }
```

Figure 9.1: HelloWorld with Remote Procedure Call

---

[1] The parameter type may be any of U, const U&, or U&&, where U is deserialized_type_t<T>, as all three parameter types accept an argument value of type U&&. The parameter type may not be U&, as that does not accept an argument value of type U&&. The parameter types const U& and U&& are recommended over U to minimize copy/move costs at invocation.

9.3 API Reference

9.3.1 One-way RPC

```
1  template<typename Func, typename ...Args>
   void rpc_ff(intrank_t recipient,
               Func &&func, Args &&...args);
   template<typename Cx, typename Func, typename ...Args>
   RType rpc_ff(intrank_t recipient,
                Cx &&completions,
                Func &&func, Args &&...args);
   template<typename Func, typename ...Args>
   void rpc_ff(const team &team, intrank_t recipient,
               Func &&func, Args &&...args);
   template<typename Cx, typename Func, typename ...Args>
   RType rpc_ff(const team &team, intrank_t recipient,
                Cx &&completions,
                Func &&func, Args &&...args);
```

Preconditions:

2   • `Func` must be a function-object type.

3   • `Func` must either be Serializable or one of the FunctionObjects listed in §6.5.

4   • If `Func&&` is an rvalue-reference type, the underlying decayed type (`std::decay<Func&&>::type`) must be MoveConstructible.

5   • `deserialized_type_t<Func>` must be MoveConstructible.

6   • Each of `Args...` must be a Serializable type, or *cq* `dist_object<T>&`, or *cq* `team&`.

7   • If a type `Arg` in `Args&&...` is an rvalue-reference type, the underlying decayed type (`std::decay<Arg>::type`) must be MoveConstructible.

8   • If a type `Arg` in `Args&&...` is not *cq* `dist_object<T>&` or *cq* `team&`, `deserialized_type_t<Arg>` must be MoveConstructible.

9   • The invocation of the deserialized function object on the deserialized arguments must not throw an exception.

10   • In the third and fourth variants, `team.is_active()` must be true.

11   In all variants, the `func` and `args...` are serialized and internally buffered before the call returns, regardless of what source-completion notifications are requested. In other words, the source-completion event always occurs before the invocation of `rpc_ff` returns. However, source-completion notifications (signaling a future, executing an LPC, or fulfilling a promise) are delayed until the next user-level progress. Requesting a notification other than buffered or blocking for source completion is deprecated, and it may be prohibited in subsequent revisions.

12   The call `rpc_ff(rank, func, args...)` is equivalent to:

```
rpc_ff(rank,
       source_cx::as_buffered(),
       func, args...)
```

13   Similarly, the call `rpc_ff(team, rank, func, args...)` is equivalent to

```
rpc_ff(team, rank,
       source_cx::as_buffered(),
       func, args...)
```

14   In the first two variants, the target of the RPC is the process whose rank is `recipient` in the world team (Ch. 11). In the latter two variants, the target is the process whose rank is `recipient` relative to the the given team.

15   After their receipt on the target, the function object and arguments are deserialized and the invocation of the deserialized function object on the deserialized arguments is enlisted for execution during user-level progress of the master persona. So long as the sending persona continues to make internal-level progress it is guaranteed that the message will eventually arrive at the recipient. See §10.5.3 `progress_required` for an understanding of how much internal-progress is necessary.

16   The result from invoking the function object is discarded. However, the return value affects the lifetime of the deserialized objects that the `UPC++` runtime constructs[2] and passes to the function object. If the return value is a non-future value or a ready future, the deserialized objects are destructed immediately after the invocation returns. On the other hand, if the return value is a non-ready future, that future must eventually be readied by a thread holding the master persona in its active stack and destruction of the deserialized objects is

---

[2]This excludes `team&` and `dist_object<T>&` arguments, where the underlying objects are not constructed by the `UPC++` runtime in deserialization.

deferred until after the future becomes ready, allowing the function object to safely initiate further asynchronous computation that operates on those objects.

17   The function object and arguments are always serialized and deserialized, even if the target is the same as the calling process. The invocation of the deserialized function object on the deserialized arguments is never performed synchronously, even if the target is the same as the calling process and `rpc_ff` is invoked during user-level progress.

18   Special handling is applied to those members of `args` which are either a reference to `dist_object` type or a `team`, as described in §6.6.

Completions:

19   • *Source*: Indicates completion of serialization of the function object and arguments.

20   *Exceptions:* The `rpc` function may throw an implementation-defined exception on the calling thread (at the initiating process) under implementation-defined conditions. The ordering of any such exception throw with respect to argument serialization and/or deserialization is unspecified. However a call throwing such an exception shall not deliver any event notifications, nor shall it lead to invocation of the function object.

21   *C++ memory ordering:* All evaluations *sequenced-before* this call will have a *happens-before* relationship with the source-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment) and the recipient's invocation of the function object.

22   *UPC++ progress level:* `internal`

9.3.2   Round-trip RPC

```
1  template<typename Func, typename ...Args>
   RType rpc(intrank_t recipient,
             Func &&func, Args &&...args);
   template<typename Cx, typename Func, typename ...Args>
   RType rpc(intrank_t recipient,
             Cx &&completions,
             Func &&func, Args &&...args);
   template<typename Func, typename ...Args>
   RType rpc(const team &team, intrank_t recipient,
             Func &&func, Args &&...args);
```

```
template<typename Cx, typename Func, typename ...Args>
RType rpc(const team &team, intrank_t recipient,
          Cx &&completions,
          Func &&func, Args &&...args);
```

Preconditions:

2   • Func must be a function-object type.

3   • Func must either be Serializable or one of the FunctionObjects listed in
           §6.5.

4   • If Func&& is an rvalue-reference type, the underlying decayed type
           (std::decay<Func&&>::type) must be MoveConstructible.

5   • deserialized_type_t<Func> must be MoveConstructible.

6   • Each of Args... must be a Serializable type, or *cq* dist_object<T>&, or
           cq team&.

7   • If a type Arg in Args&&... is an rvalue-reference type, the underlying
           decayed type (std::decay<Arg>::type) must be MoveConstructible.

8   • If a type Arg in Args&&... is not *cq* dist_object<T>& or *cq* team&,
           deserialized_type_t<Arg> must be MoveConstructible.

9   • The result of applying the deserialized function object to the dese-
           rialized arguments must either be of a Serializable type that is not
           view<U, IterType>, or it must be future<T...>, where each type in
           T... must be Serializable but not view<U, IterType>. In either case,
           the deserialized types corresponding to these Serializable types must be
           MoveConstructible.

10   • If the return type RetType of the function object is of non-
           reference or rvalue-reference type, the underlying decayed type
           (std::decay<RetType>::type) must be MoveConstructible.

11   • The invocation of the deserialized function object on the deserialized ar-
           guments must not throw an exception.

12   • In the third and fourth variants, team.is_active() must be true.

13   Similar to rpc_ff, this call sends func and args... to be executed remotely,
      but additionally provides an operation-completion event. This event produces
      the value returned from the remote invocation of the deserialized function object
      on its deserialized arguments, if it is non-void.

14   In all variants, the `func` and `args...` are serialized and internally buffered before the call returns, regardless of what source-completion notifications are requested. In other words, the source-completion event always occurs before the invocation of `rpc` returns. However, source-completion notifications (signaling a future, executing an LPC, or fulfilling a promise) are delayed until the next user-level progress. Requesting a notification other than buffered or blocking for source completion is deprecated, and it may be prohibited in subsequent revisions.

15   The call `rpc(rank, func, args...)` is equivalent to:

```
rpc(rank,
    source_cx::as_buffered() | operation_cx::as_future(),
    func, args...)
```

16   Similarly, the call `rpc(team, rank, func, args...)` is equivalent to:

```
rpc(team, rank,
    source_cx::as_buffered() | operation_cx::as_future(),
    func, args...)
```

17   In the first two variants, the target of the RPC is the process whose rank is `recipient` in the world team (Ch. 11). In the latter two variants, the target is the process whose rank is `recipient` relative to the the given team.

18   After their receipt on the target, the function object and arguments are deserialized and the invocation of the deserialized function object on the deserialized arguments is enlisted for execution during user-level progress of the master persona.

19   In the first variant, the returned future is readied upon operation completion.

20   For futures provided by an operation-completion request, or promises used in promise-based operation-completion requests, the type of the future or promise must correspond to the return type of the invocation of the function object as follows:

21   • If the return type is of the form `future<T...>`, then a future provided by operation completion has type `future<deserialized_type_t<T>...>`, and promises used in operation-completion requests must permit invocation of `fulfill_result` with values of type `deserialized_type_t<T>...`.

22   • If the return type is some other non-`void` type `T`, then a future provided by operation completion has type `future<deserialized_type_t<T>>`, and

promises used in operation-completion requests must permit invocation of `fulfill_result` with a value of type `deserialized_type_t<T>`.

23
- If the return type is `void`, then a future provided by operation completion has type `future<>`, and promises used in operation-completion requests may have any type `promise<T...>`.

24 Within user-progress of the recipient's master persona, the result from invoking the function object will be immediately serialized if the result is a non-future value or a ready future. If the result is a non-ready future, that future must eventually be readied by a thread holding the master persona in its active stack and the encapsulated value will be serialized when the future becomes ready. In both cases, the deserialized objects that the `UPC++` runtime constructs[3] and passes to the function object are destructed after serialization of the result is complete. This allows the function object to safely initiate further asynchronous computation that operates on the deserialized objects or return a result that contain references to those objects. The serialized value is eventually sent back to the initiating process. Upon receipt, it will be deserialized, and operation-completion notifications will take place during subsequent user-progress of the initiating persona.

25 The function object and arguments are always serialized and deserialized, even if the target is the same as the calling process. The invocation of the deserialized function object on the deserialized arguments is never performed synchronously, even if the target is the same as the calling process and `rpc` is invoked during user-level progress.

26 The same special handling applied to `dist_object&` and `team&` arguments by `rpc_ff` is also done by `rpc`.

Completions:

27
- *Source*: Indicates completion of serialization of the function object and arguments.

28
- *Operation*: Indicates completion of all aspects of the operation: serialization, deserialization, remote invocation, transfer of any result, and destruction of any internally managed values are complete. This completion produces a value as described above.

---

[3]This excludes `team&` and `dist_object<T>&` arguments, where the underlying objects are not constructed by the `UPC++` runtime in deserialization.

²⁹ *Exceptions:* The `rpc_ff` function may throw an implementation-defined exception on the calling thread (at the initiating process) under implementation-defined conditions. The ordering of any such exception throw with respect to argument serialization and/or deserialization is unspecified. However a call throwing such an exception shall not deliver any event notifications, nor shall it lead to invocation of the function object.

³⁰ *C++ memory ordering:* All evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of the function object. The return from the invocation of the function object will have a *happens-before* relationship with the operation-completion actions (future readying, promise fulfillment, or persona LPC enlistment). For LPC completions, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of the completion function.

³¹ *UPC++ progress level:* `internal`

Chapter 10

Progress

10.1 Overview

1 UPC++ presents a highly-asynchronous interface, but guarantees that user-provided callbacks will only ever run on user threads during calls to the library. This guarantees a good user-visibility of the resource requirements of UPC++, while providing a better interoperability with other software packages which may have restrictive threading requirements. However, such a design choice requires the application developer to be conscientious about providing UPC++ access to CPU cycles.

2 Progress in UPC++ refers to how the calling application allows the UPC++ internal runtime to advance the state of its outstanding asynchronous operations. Any asynchronous operation initiated by the user may require the application to give UPC++ access to the execution thread periodically until the operation reports its completion. Such access is granted by simply making calls into UPC++. Each UPC++ function's contract to the user contains its *progress guarantee* level. This is described by the members of the upcxx::progress_level enumerated type:

3 `progress_level::user` UPC++ may advance its internal state as well as signal completion of user-initiated operations. This may entail the firing of remotely injected procedure calls (RPCs), or readying/fulfillment of futures/promises and the ensuing callback cascade.

4 `progress_level::internal` UPC++ may advance its internal state, but no notifications will be delivered to the application other than eager notification (Ch. 7) of events associated with the invoked function itself. Thus, an application has very limited ways to "observe" the effects of such progress.

5 *Progress level: none* `UPC++` will not attempt to advance the progress of asynchronous operations. (Note this level does not have an explicit entry in the `progress_level` enumerated type).

6 The most common progress guarantee made by `UPC++` functions is *Progress level: none.* As indicated in §1.6, all `UPC++` functions implicitly guarantee *Progress level: none*, unless explicitly specified otherwise. Calls to functions with *Progress level: none* will never progress `UPC++` internal state nor execute user callbacks.

7 Most functions that inject communication have `progress_level::internal`. This ensures the delivery of notifications to remote processes (or other threads) making *user*-level progress in a timely manner. The invocation of a function with `progress_level::internal` will notably never execute callbacks or RPCs before returning. In order to avoid having the user contend with the cost associated with callbacks and RPCs being run anytime a `UPC++` function is entered, `progress_level::user` is purposefully not the common case.

8 `progress` is the notable function enabling the application to make *user*-level progress. Its sole purpose is to look for ready operations involving this process or thread and run the associated RPC/callback code:

```
upcxx::progress(progress_level lev = progress_level::user)
```

9 `UPC++` execution phases which leverage asynchrony heavily tend to follow a particular program structure. First, initial communications are launched. Their completion callbacks might then perform a mixture of compute or further `UPC++` communication with similar, cascading completion callbacks. Then, the application spins on `upcxx::progress()`, checking some designated application state which monitors the amount of pending outgoing/incoming/local work to be done. For the user, understanding which functions perform these progress spins becomes crucial, since any invocation of user-level progress may execute RPCs or callbacks.

10.2   Restricted Context

1 During user-level progress made by `UPC++`, callbacks may be executed. Such callbacks are subject to restrictions on how they may further invoke `UPC++` themselves. We designate such restricted execution of callbacks as being in the *restricted context*. The general restriction is stated as:

2   *User code running in the restricted context must assume that for the duration of the context all other attempts at making progress, from any thread, may result in a no-op every time.*

3   The immediate implication is that a thread which is already in the restricted context should assume no-op behavior from further attempts at making progress. This makes it pointless to try and wait for `UPC++` notifications from within the restricted context, since there is no viable mechanism to make such notifications visible to the user. In particular, attempts to spin on progress while awaiting delivery of a `UPC++` notification from within the restricted context will likely hang the thread[1].

4   A thread running in the restricted context shall not initiate any `UPC++` collective operation (§12).

5   The `in_progress` function can be used to query whether the calling thread is currently running in the restricted context.

10.3   Attentiveness

1   Many `UPC++` operations have a mechanism to signal completion to the application. However, a performance-oriented application will need to be aware of an additional asynchronous operation status indicator called *progress-required*. This status indicates that for a particular operation further advancements of the current process or thread's *internal-*level progress are necessary so that completion regarding remote entities (e.g. notification of delivery) can be reached. Once an operation has left the progress-required state, `UPC++` guarantees that remote entities will see their side of the operations' completion without any further progress by the current compute resource. Applications will need to leverage this information for performance, as it is inadvisable for a compute resource to become inattentive to `UPC++` progress (e.g. long bouts of arithmetic-heavy computation) while other entities depend on operations that require further servicing.

2   As said previously, nearly all `UPC++` operations track their completion individually. However, it is not possible for the programmer to query `UPC++` if individual operations no longer require further progress. Instead, the user may ask `UPC++` when operations initiated by this thread have reached a state at which they no longer require internal progress to reach their destinations. So for example, one may ask whether `rpc` or `rput` operations previously initiated by this thread and destined for a remote process have been handed off to the network hardware (but not necessarily delivered/completed).

---

[1]For this reason, some blocking calls such as `future<T...>::wait()` are prohibited within the restricted context. A common idiom for this use case is to instead schedule a completion via `future<T...>::then()`.

3 This is achieved by using the following functions:

```
bool upcxx::progress_required();
void upcxx::discharge();
```

4 The `progress_required` function reports whether this thread requires internal progress on outgoing operations, allowing the application to know that there are still pending outgoing operations that will not achieve remote completion without further advancements to internal progress. This is of particular importance before a thread enters a lapse of inattentiveness (for instance, performing expensive computations) in order to prevent slowing down remote entities.

5 The `discharge` function allows a thread to ensure that `UPC++` no longer requires internal progress to deliver operations outgoing from this thread[2]. It is equivalent to the following:

```
6 void upcxx::discharge(persona_scope &ps = default_persona_scope()){
     assert(!upcxx::in_progress()); // prohibit restricted context
     while(upcxx::progress_required(ps))
       upcxx::progress(upcxx::progress_level::internal);
   }
```

7 A well-behaved `UPC++` application is encouraged to call `discharge` before any long lapse of attentiveness to progress.

10.4   Thread Personas/Notification Affinity

1 As explained in Chapter 5 *Futures and Promises*, futures require careful consideration when used in the presence of thread concurrency. It is crucial that `UPC++` is very explicit about how a multi-threaded application can safely use futures returned by `UPC++` calls.

2 The notion of "thread" has been used in a loose fashion throughout portions of this document, the natural interpretation being an operating system (OS) thread. More precisely, this document often uses the term "thread" to denote a `UPC++` abstraction referred to as *thread persona* (or simply "persona") which generalizes the notion of an OS thread.

3 A `UPC++` thread persona is a collection of `UPC++`-internal state usually attributed to a single thread. By making it a proper construct, `UPC++` allows a single OS thread to switch between multiple application-defined roles for processing notifications. Personas act as the receivers for notifications generated by the `UPC++` runtime.

---

[2]Actually, this only applies to non-master personas active with this thread; see the API reference for detailed semantics. Personas are discussed in the next section.

4   Values of type `upcxx::persona` are non-copyable, non-movable objects which the application can instantiate as desired. For each OS thread, `UPC++` internally maintains a stack of *active* persona references. The top of this stack is the *current* persona. All asynchronous `UPC++` operations will have their notification events (signaling of futures or promises) sent to the current persona of the OS thread invoking the operation. Calls that make user-level progress will process notifications destined to any of the active personas of the invoking thread. For the duration of a notification's processing, its target persona is placed at the top of the persona stack of the OS thread associated with that persona.

5   The most important implication for application code is that event completions are reported to a specific persona, regardless of which thread currently holds that persona. For example, when a thread invokes an asynchronous `UPC++` operation that returns a `future`, that future becomes implicitly associated with the current persona of the calling thread. After operation completion, that future will later be readied during user-level progress of that same persona (regardless of which thread currently holds it). In the common case where this persona does not migrate across threads, that means the completion notification is delivered to the same thread which initiated the operation. However if the application explicitly transfers the persona to a different thread while the operation is in-flight, the completion will instead be delivered during progress of the thread now holding the associated persona. An example of this pattern will be described later in this section.

6   The initial state of the persona stack for each thread consists of a single entry pointing to a persona known as the *default persona*. A default persona is created automatically by `UPC++` for each OS thread and remains pinned to that OS thread. The default persona may never be removed from the persona stack or added to the persona stack of a different thread.

7   There is one special persona per process, the *master persona*, which is created automatically by `UPC++` and pushed onto the active persona stack of the thread which initializes the library by calling `upcxx::init()` (§2). The master persona is special in that it is the only persona in each process that can execute incoming RPC callbacks. The master persona is also a specified precondition for invoking most collective operations and certain other operations.

8   Pushing and popping personas from the persona stack (hence changing the current persona) is done with the `upcxx::persona_scope` type. For example:

```
1 persona scheduler_persona;
2 std::mutex scheduler_lock;
3
4 { // Scope block delimits domain of persona_scope instance.
5   auto scope = persona_scope(scheduler_lock, scheduler_persona);
6
7   // All following upcxx actions will use 'scheduler_persona'
```

```
8    // as current.

9

10   // ...

11

12   // `scope` destructs:
13   // - `scheduler_persona` dropped from active set if it
14   //   wasn't active before the scope's construction.
15   // - Previously current persona revived.
16   // - Lock released.
17 }
```

9 Since `UPC++` will assume an OS thread has exclusive access to all of its active personas, it is the user's responsibility to ensure that no OS threads share an active persona concurrently. The use of the `persona_scope` constructor, which takes a lock-like synchronization primitive, is strongly encouraged to facilitate in enforcing this invariant.

10 To summarize, each `future` returned by a `UPC++` asynchronous operation is implicitly associated with a particular `persona`, each `persona` may only be active with a single OS thread at a time, and the future will only be readied during user-level progress of the thread where the persona is active.

11 There are two ways that asynchronous operations can be initiated by a given OS thread but retired in another. The first solution is simple:

1. The user defines a persona `P`.

2. Thread 1 activates `P`, initiates the asynchronous operation, and releases `P`.

3. Thread 1 synchronizes with Thread 2, indicating the operation has been initiated.

4. Thread 2 activates `P`, spins on `progress` until the operation completes.

12 Care must be taken that any futures created by phase 2 are never altered (uttered) concurrently. The same synchronization that was used to enforce exclusivity of persona acquisition can be leveraged to protect the future as well.

13 While this technique achieves our goal of different threads initiating and resolving asynchronous operations, it fails a different but also desirable property. It is often desirable to allow multiple threads to issue communication *concurrently* while delegating a separate thread to handle the notifications. To achieve this, it is clear that multiple personas are needed. Indeed, the exclusivity of a persona being current to only one OS thread prevents the application from concurrent initiation of communication.

14 In order to issue operations and concurrently retire them in a different thread, the user is strongly encouraged to use the LPC completion mechanism described in Chapter 7, as opposed to the future or promise variants. An example of such a call is:

15
```
rget(gptr_src, operation_cx::as_lpc(some_persona, callback_func));
```

16 In addition to the arguments necessary for the particular operation, the `as_lpc` completion mechanism takes a reference to a persona (which need not be active with the calling thread) and a `C++` function object (lambda, etc.). Upon completion of the operation, the designated persona shall execute the function object during its user-level progress. Using this mechanism, it is simple to have multiple threads initiating communication concurrently with a designated thread receiving the completion notifications. To achieve this, each operation is initiated by a thread using the agreed-upon persona of the receiver thread together with a callback that will incorporate knowledge of completion into the receiver's state.

10.5   API Reference

1
```
enum class progress_level {
  /*none, -- not an actual member, conceptual only*/
  internal,
  user
};
```

2
```
void progress(progress_level lev = progress_level::user);
```

3   This call will always attempt to advance internal progress.

4   If `lev == progress_level::user` then this call may execute (in an unspecified order) available user actions for any personas active with the calling thread. Actions include:

   1. Either future-readying or promise-fulfilling completion notifications for asynchronous operations initiated by one of the active personas. By the execution model of futures and promises this can induce callback cascade.

   2. Continuation-style completion notifications from operations initiated by any persona but designating one of the active personas as the completion recipient.

    3. RPCs (§9) destined for this process, if and only if the master persona is among the active set.

    4. LPC's destined for any of the active personas.

5   *UPC++ progress level:* `internal` or `user`

6
```
bool in_progress();
```

7   Returns true if and only if the calling thread is currently executing in the restricted context (§10.2), in other words, within the dynamic scope of user-level progress.

10.5.1   persona

1
```
class persona;
```

2   *C++ Concepts:* DefaultConstructible, Destructible

3
```
persona::persona();
```

4   Constructs a persona object with no enqueued operations.

5   *This function may be called when* `UPC++` *is in the uninitialized state.*

6
```
persona::~persona();
```

7   Destructs this persona object. If this persona is a member of any thread's persona stack, the result of this call is undefined. If any operations are currently enqueued on this persona, or if any operations initiated by this persona require further progress, the result of this call is undefined.

8   *This function may be called when* `UPC++` *is in the uninitialized state.*

```
9 template < typename Func >
  void persona :: lpc_ff ( Func && func );
```

Preconditions:

10 • `Func` must be a function-object type that can be invoked on zero arguments.

11 • If `Func&&` is an rvalue-reference type, the underlying decayed type (`std::decay<Func&&>::type`) must be MoveConstructible.

12 • If `Func&&` is an lvalue-reference type, the underlying decayed type (`std::decay<Func&&>::type`) must be CopyConstructible.

13 • The invocation of the function `func()` must not throw an exception.

14 `std::forward`'s `func` into the construction of a temporary function object `tempfunc` which is enlisted for invocation during user-level progress of the target persona (`this`), where it will be invoked in a manner equivalent to `std::move(tempfunc)()`.

15 The return value of the invocation (if any) is ignored, and the temporary function object is destroyed by the target persona after the invocation returns.

16 This function is thread-safe, so it may be called from any thread to enqueue work for this persona.

17 The invocation of the function object is never performed synchronously during the call to `lpc_ff`, even if the target persona is a member of the caller's persona stack and this function is invoked during user-level progress.

18 *C++ memory ordering:* All evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of the function object.

```
19 template < typename Func >
  FType persona :: lpc ( Func && func );
```

Preconditions:

20 • `Func` must be a function-object type that can be invoked on zero arguments.

21 • If `Func&&` is an rvalue-reference type, the underlying decayed type (`std::decay<Func&&>::type`) must be MoveConstructible.

22 • If `Func&&` is an lvalue-reference type, the underlying decayed type (`std::decay<Func&&>::type`) must be CopyConstructible.

23   • The invocation of the function `func()` must not throw an exception.

24   • If the return type `RetType` of Func is of non-reference or rvalue-reference type, the underlying decayed type (`std::decay<RetType>::type`) must be MoveConstructible.

25   • If the return type of Func is of the form `future<T...>`, then each non-reference type in `T...` must be MoveConstructible.

26   `std::forward`'s func into the construction of a temporary function object `tempfunc` which is enlisted for invocation during user-level progress of the target persona (`this`), where it will be invoked in a manner equivalent to `std::move(tempfunc)()`. The temporary function object is destroyed by the target persona after the invocation returns.

27   Returns a new future representing the return value resulting from the asynchronous invocation of the function object by the target persona (`this`). The return type `FType` and semantics are as follows:

28   • If the function object invocation returns a future type `future<U...>`, then `FType` is also `future<U...>`. The future returned by `lpc` encapsulates the same set of results as the future returned by the function object invocation. The future returned by `lpc` is readied during user-level progress of the calling persona after both the invocation of the function object has completed and the future returned by that invocation has been readied.

29   • If the function object invocation returns a non-future, non-`void` type U, then `FType` is determined as follows: If U is an lvalue reference, then `FType` is `future<U>`. Otherwise, `FType` is `future<typename std::decay<U>::type>`. The future returned by `lpc` encapsulates the result returned by the function object invocation. The future returned by `lpc` is readied during user-level progress of the calling persona after the invocation of the function object has completed.

30   • If `func` has `void` return type, then `FType` is `future<>`. The future returned by `lpc` represents whether or not the invocation of the function object has completed. The future returned by `lpc` is readied during user-level progress of the calling persona after the invocation of the function object has completed.

31   This function is thread-safe, so it may be called from any thread to enqueue work for this persona. The future returned by `lpc` is associated with the current persona of the caller. Conversely, any future returned by the function object invocation must have been constructed by the target persona (`this` was the

current persona for the thread constructing the future). Furthermore, if the returned future is non-ready, then it must eventually be readied by a thread holding the target persona in its active stack (§5.7).

32   The invocation of the function object is never performed synchronously during the call to `lpc`, even if the target persona is a member of the caller's persona stack and this function is invoked during user-level progress.

33   *C++ memory ordering:* All evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of the function object, and the invocation of the function object will have a *happens-before* relationship with evaluations sequenced after the signaling of the final future.

34 `bool persona::active_with_caller() const;`

35   Returns true if and only if this persona is a member of the calling OS thread's persona stack.

36 `persona& master_persona();`

37   Returns a reference to the master persona automatically instantiated by the `UPC++` runtime. The thread that executes `upcxx::init` implicitly acquires this persona as its current persona. The master persona is special in that it is the only one which will execute RPCs destined for this process. Additionally, some `UPC++` functions may only be called by a thread with the master persona in its active stack.

38 `persona& current_persona();`

39   Returns a reference to the persona on the top of the thread's active persona stack.

40 `persona& default_persona();`

41   Returns a reference to the persona instantiated automatically and uniquely for this OS thread. The default persona is always the bottom of and can never be removed from its designated OS thread's active stack.

42 `void liberate_master_persona()`

43   *Precondition:* This thread must be the one which called `upcxx::init`, it must have not altered its persona stack since calling `init`, and it must not have called this function already since calling `init`.

44   The thread which invokes `upcxx::init` implicitly has the master persona at the top of its active stack, yet the user has no `persona_scope` to drop to allow other threads to acquire the persona. Thus, if the user intends for other threads to acquire the master persona, they should have the init-calling thread release the persona with this function so that it can be claimed by `persona_scope`'s. Generally, if this function is ever called, it is done soon after `init` and then the master persona should be reacquired by a `persona_scope`.

10.5.2   persona_scope

1 `class persona_scope;`

2   *C++ Concepts:* Destructible, MoveConstructible

3 `persona_scope::persona_scope(persona &p);`

4   *Precondition:* Excluding this thread, `p` is not a member of any other thread's active stack.

5   Pushes `p` onto the top of the calling OS thread's active persona stack.

6 `template<typename Mutex>`
`persona_scope::persona_scope(Mutex &mutex, persona &p);`

7   *C++ Concepts of `Mutex`:* Mutex

8   *Precondition:* `p` will only be a member of some thread's active stack if that thread holds `mutex` in a locked state.

9   Invokes `mutex.lock()`, then pushes `p` onto the OS thread's active persona stack.

```
10 persona_scope::~persona_scope();
```

11   *Precondition:* All `persona_scope`'s constructed on this thread since the construction of this instance have since destructed.

12   The persona supplied to this instance's constructor is popped from this thread's active stack. If this instance was constructed with the mutex constructor, then that mutex is unlocked.

13   *This function may be called when `UPC++` is in the uninitialized state.*

```
14 persona_scope& top_persona_scope();
```

15   Reference to the most recently constructed but not destructed `persona_scope` for this thread. Every thread begins with an implicitly instantiated scope pointing to its default persona that survives for the duration of the thread's lifetime.

```
16 persona_scope& default_persona_scope();
```

17   Every thread begins with an implicitly instantiated scope pointing to its default persona that survives for the duration of the thread's lifetime. This function returns a reference to that bottommost `persona_scope` for the calling thread, which points at the calling thread's `default_persona()`.

10.5.3   Outgoing Progress

```
1 bool progress_required(persona_scope &ps =
                            default_persona_scope());
```

2   *Precondition:* `ps` has been constructed by this thread.

3   For the set of personas included in this thread's active stack section bounded inclusively between `ps` and the current top, *nearly* answers if any `UPC++` operations initiated by those personas require further advancement of internal-progress of their respective personas before their completion events will be eventually available to user-level progress on the destined processes. The exact meaning of the return value depends on which personas are selected by `ps`:

4   • If `ps` *does not* include the master persona: A return value of `true` means that one or more of the personas indicated by `ps` requires further internal-progress to achieve completion of its outgoing operations. A value of `false` means that none of the personas indicated by `ps` require internal-progress, but internal-progress of the master persona might still be required.

5   • If `ps` *does* include the master persona: A return value of `true` means that one or more of the personas indicated by `ps` requires further internal-progress to achieve completion of its outgoing operations. A return value of `false` means that none of the non-master personas indicated by `ps` requires further internal-progress, but the master persona may or may not require further internal-progress.

6 ```
void discharge(persona_scope &ps = default_persona_scope());
```

7   *Precondition:* `ps` has been constructed by this thread.

8   Advances internal-progress sufficiently to ensure that `progress_required(ps)` returns `false`.

9   *Note*: `discharge()` only ensures that internal progress has been advanced sufficiently to guarantee that *outgoing* operations initiated by the selected *non-master* personas active with this thread will eventually reach their destinations. In particular, it does not guarantee anything about communication arrival at other processes or acknowledgements to those operations, for example acknowledgements that trigger operation completion events.

10   *This function may not be invoked from the restricted context (§10.2).*

11   *UPC++ progress level:* `internal`

Chapter 11

Teams

11.1 Overview

1   UPC++ provides *teams* as a means of grouping processes. UPC++ uses `team`s for collective operations (Ch. 12). `team` construction is collective and should be considered moderately expensive and done as part of the set-up phase of a calculation. `team`s are similar to `MPI_Group`s and the default `team` is `world()`. `team`s are considered special when it comes to serialization. Each `team` has a unique `team_id` that is equal across the `team` and acts as an opaque handle. Any process that is a member of the `team` can retrieve the `team` object with the `team_id::here()` function. Hence, coordinating processes can reference specific `team`s by their `team_id`.

2   While a process within a `UPC++` SPMD program can have multiple `intrank_t` values that represent their relative placement in several `team`s, it is the `intrank_t` in the `world()` team that is used in most `UPC++` functions, unless otherwise specifically noted. For example, `broadcast` takes a rank relative to the specific team over which it operates.

11.2 Local Teams

1   The local team is an ordered set of processes where heap storage in the shared segment allocated by any process in the team is local to all members. Any process can obtain a reference to the local team by calling `local_team` and global pointers behave accordingly:

1. `global_ptr`'s referencing objects allocated in the shared segment of processes that are members of this `team` will report `is_local() == true` and `local()` will return a valid `T*` referencing the corresponding object.

2. The `global_ptr where()` function will report the rank in team `world()` of the process that originally acquired the referenced object using the functions in chapter 4.

2 It is *not* guaranteed that the `T*`'s obtained by different processes to the same shared object will have bit-wise identical pointer values. In the general case, peers may have different virtual addresses for the same physical memory.

11.3 API Reference

11.3.1 team

1 ```
class team final;
```

2   *C++ Concepts:* DefaultConstructible, MoveConstructible, MoveAssignable, Destructible

3 ```
constexpr intrank_t team::color_none;
```

4   A constant used to specify that the calling process of `split()` will not be a member of any subteam. This constant is guaranteed to have a negative value.

5 ```
bool team::is_active() const;
```

6   Returns whether or not this `team` is active. A `team` is active if both the following hold:

7   • It was created as a fundamental team, via `team::split()` with a `color` argument other than `team::color_none`, via `team::create()` with a non-empty sequence, or as the target of move construction or move assignment from an active team.

8   • It has not subsequently been passed as an argument to the move constructor or move assignment operator, nor had its state destroyed by a call to `team::destroy()`.

<sup>9</sup> `intrank_t team::rank_n() const;`

<sup>10</sup>     *Precondition:* `this->is_active()`.

<sup>11</sup>     Returns the number of ranks in the given team.

<sup>12</sup> `intrank_t team::rank_me() const;`

<sup>13</sup>     *Precondition:* `this->is_active()`.

<sup>14</sup>     Returns the rank of the calling process in the given team.

<sup>15</sup> `intrank_t team::operator[](intrank_t peer_index) const;`

<sup>16</sup>     *Precondition:* `this->is_active()`. `peer_index >= 0` and `peer_index < rank_n()`.

<sup>17</sup>     Returns the rank in the `world()` team of the process with rank `peer_index` in this team.

<sup>18</sup>     *UPC++ progress level:* unspecified between `none` and `internal`

<sup>19</sup> `intrank_t team::from_world(intrank_t world_index) const;`
`intrank_t team::from_world(intrank_t world_index,`
`                            intrank_t otherwise) const;`

<sup>20</sup>     *Precondition:* `this->is_active()`. `world_index >= 0` and `world_index < world().rank_n()`. For the single-argument overload, the process with rank `world_index` in the `world()` team must be a member of this team.

<sup>21</sup>     Returns the rank in this team of the process with rank `world_index` in the `world()` team. For the two-argument overload, if that process is not a member of this team then the value of `otherwise` is returned.

<sup>22</sup>     *UPC++ progress level:* unspecified between `none` and `internal`

```
23  team team::split(intrank_t color, intrank_t key) const;
```

24   *This function is collective (§12.1) over this (i.e. the parent) team, the team must be active, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

25   *Precondition:* `color >= 0 || color == team::color_none`

26   Splits the given team into subteams based on the `color` and `key` arguments.

27   If `color == team::color_none`, the return value is an inactive team.

28   Otherwise, all processes that call the function with the same `color` value will be separated into the same subteam. Ranks in the same subteam will be numbered according to their position in the sequence of sorted key values. If two callers specify the same combination of `color` and `key`, their relative ordering in the subteam will be the same as in the parent team. The return value is the `team` representing the calling process's new subteam, and the resulting `team` is active.

29   *C++ memory ordering:* With respect to all threads participating in this collective and passing a `color` that is not `team::color_none`, all evaluations which are *sequenced-before* their respective thread's invocation of this call will have a *happens-before* relationship with all evaluations sequenced after the call.

30   *UPC++ progress level:* `user`

```
31  template<typename Iter>
    team team::create(Iter begin, Iter end) const;
    template<typename Iter>
    team team::create(Iter begin, Iter end, size_t count) const;
    template<typename Container>
    team team::create(const Container &c) const;
```

32   *This function is collective (§12.1) over this (i.e. the parent) team, the team must be active, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

33   *Precondition:* In the first and second variants, `Iter` must satisfy the InputIterator C++ concept and `*std::declval<Iter>()` must have a type convertible to `intrank_t`. In the second variant `count` must be equal to the number of elements in `[begin, end)`. In the third variant, `Container` must satisfy the Container C++ concept[1] and the underlying element type `Container::value_type` must be convertible to `intrank_t`. In all cases, if the input sequence of `intrank_t` values is non-empty, then all values must be distinct integers in the range `[0..this->rank_n())`, and the sequence must contain `this->rank_me()`.

34   Splits the parent team into zero or more disjoint subset teams. The return value is the `team` representing the calling process's new subteam.

35   Each process passes a sequence of ranks in `this` (i.e. the parent) team which specifies the membership of the new subteam it will join. If a process passes a non-empty sequence of ranks, then all processes identified by those ranks must pass the same sequence, and the resulting `team` is active.

36   Any process passing an empty sequence does not become a member of a new subteam, and the return value to such callers is an inactive `team`.

37   *Advice to users:* `team::create` can express the same team subdivision operations as `team::split`. The former requires all participants to directly provide the sequence of ranks comprising their new team, whereas the latter must construct this information dynamically via interprocess communication. In cases where all participants can independently compute this sequence, `create` may require less communication and synchronization than an equivalent `split` call.

38   *C++ memory ordering:* Although this is a collective call, it does not provide any synchronization or ordering guarantees between participating threads.

39   *UPC++ progress level:* `user`

---

[1]Note the Container iterators must traverse the elements in a deterministic order to satisfy the constraint on sequence equality across processes.

40 `team :: team ();`

41   Constructs an inactive team.

42   *This function may be called when* `UPC++` *is in the uninitialized state.*

43 `team :: team ( team && other );`

44   *Precondition:* The master persona (§10.5.1) must appear in the persona stack of the calling thread. `other` must not reference the `world()` or local team. No operation on the team associated with `other`, nor any `UPC++` operation with a progress level other than `none`, may have been invoked by the calling process between the creation of `other` and this call.

45   Makes this instance the calling process's representative of the team associated with `other`, transferring all state from `other`. Deactivates `other`.

46 `team& team :: operator =( team && other );`

47   *Precondition:* `!this->is_active()`. The master persona (§10.5.1) must appear in the persona stack of the calling thread. `other` must not reference the `world()` or local team. No operation on the team associated with `other`, nor any `UPC++` operation with a progress level other than `none`, may have been invoked by the calling process between the creation of `other` and this call.

48   Makes this instance the calling process's representative of the team associated with `other`, transferring all state from `other`. Deactivates `other`.

49 `void team :: destroy ( entry_barrier lev = entry_barrier :: user );`

50   If `this` references an inactive team, then `this->destroy()` is a non-collective call with no preconditions, progress level `none` and no other semantics (i.e., the call has no effect). Otherwise, the following specifications apply.

51   *This function is collective (§12.1) over this team, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

52   *Precondition:* `this` must not reference the `world()` or local team. `lev` must be single-valued (Ch. 12). After the entry barrier (§12.2) specified by `lev` completes, or upon entry if `lev == entry_barrier::none`, the operations on this team must not require internal-level or user-level progress from any persona before they can complete.

53   Destroys the calling process's state associated with the team. Further lookups on this process using the `team_id` corresponding to this team will have undefined behavior. Deactivates this team object.

54   *C++ memory ordering:* If `lev != entry_barrier::none`, with respect to all threads participating in this collective, all evaluations which are *sequenced-before* their respective thread's invocation of this call will have a *happens-before* relationship with all evaluations sequenced after the call.

55   *UPC++ progress level:* `user` if `lev == entry_barrier::user`, `internal` otherwise

56   `team::~team();`

57   *Precondition:* Either `UPC++` must have been uninitialized since the team's creation, or this must be an inactive team.

58   Destructs this `team` object.

59   *Advice to users:* `team::destroy()` should be used to deactivate an active `team` before destruction.

60   *This function may be called when `UPC++` is in the uninitialized state.*

61   `team_id team::id() const;`

62   *Precondition:* `this->is_active()`.

63   Returns the universal name that uniquely identifies this team.

11.3.2 team_id

¹ ```
class team_id;
```

² *C++ Concepts:* DefaultConstructible, TriviallyCopyable, StandardLayoutType, EqualityComparable, LessThanComparable, hashable

³ **UPC++** *Concepts:* TriviallySerializable

⁴ A universal name that uniquely identifies a team.

⁵ ```
team_id::team_id();
```

<sup>6</sup> Initializes this name to be the invalid ID. All default-constructed `team_id` objects will receive the same, unique invalid identifier. This enables use of `team_id()` as a placeholder ID for naming the absence of a team.

<sup>7</sup> ```
team& team_id::here() const;
```

⁸ *Precondition:* This name must be a valid ID. The calling process must be a member of the team associated with this name, and it must have completed creation of the team. The team must not have been destroyed.

⁹ Retrieves a reference to the `team` instance associated with this name.

¹⁰ ```
future<team &> team_id::when_here() const;
```

<sup>11</sup> *Precondition:* This name must not be the invalid ID. The corresponding team must include the calling process and must not have already been destroyed by the caller. The master persona (§10.5.1) must appear in the persona stack of the calling thread.

<sup>12</sup> Retrieves a future which is readied during user-level progress of the master persona after the corresponding team is constructed on the calling process. If the corresponding team has already been created when this is called, may return a ready future.

### 11.3.3   Fundamental Teams

1  `team& world();`

2       Returns a reference to the team representing all the processes in the `UPC++`
        program. The result is undefined if a move is performed on the returned team.
        Calling `destroy()` on the returned team results in undefined behavior.

3  `intrank_t rank_n();`

4       Returns the number of ranks in the `world()` team.

5       Equivalent to: `world().rank_n()`.

6  `intrank_t rank_me();`

7       Returns the rank of the calling process in the `world()` team.

8       Equivalent to: `world().rank_me()`.

9  `team& local_team();`

10      Returns a reference to the local team containing this process. The local team
        represents an ordered set of processes where memory allocated from the shared
        segment of any member is local to all team members (§11.2).  The result is
        undefined if a move is performed on the returned team. Calling `destroy()` on
        the returned team results in undefined behavior.

11  `bool local_team_contains(intrank_t world_index);`

12      *Precondition:* `world_index >= 0` and `world_index < world().rank_n()`.

13      Determines if the process whose rank is `world_index` in `world()` is a member
        of the local team containing the calling process (§11.2).

14      Equivalent to: `local_team().from_world(world_index,-1) >= 0`

```
15  std :: pair < intrank_t , intrank_t > local_team_position ();
```

16   Queries information about the disjoint local teams comprising `world()`.

17   Returns a value such that if the result is assigned to the `std::pair` variable
        `info`, then:

18   • `info.second` provides the number of disjoint local teams in the set com-
              prising `world()`. During a given execution, this value is equal for all
              callers.

19   • `info.first` provides an integral index in `[0,info.second)` that identifies
              the local team of the calling process within that set. During a given
              execution, the value returned to two processes is equal if and only if they
              share a local team.

20   The values returned to any given calling process remain stable across subse-
        quent calls.

21   *Advice to Users*: This function returns information about the number and iden-
        tity of local teams, which delineate the boundaries of shared heap locality within
        the job (and may correspond to physical node boundaries). Information about a
        caller's position *within* its local team is available via `local_team().rank_me()`
        and `local_team().rank_n()`.

Chapter 12

Collectives

1   A *collective operation* is a `UPC++` operation that must be matched across all participating processes. Informally, any two processes that both participate in a pair of collective operations must agree on their ordering. Furthermore, if a parameter or other property of a collective operation is specified as *single-valued*, all participating processes must provide the same value for the parameter or property.

2   A collective operation need not provide any actual synchronization between processes, unless otherwise noted. The collective requirement simply states a relative ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any valid program. Some implementations may include unspecified synchronization between processes within collective operations, but programs must not rely upon the presence or absence of such unspecified synchronization for correctness.

3   A noteworthy exception to the previous paragraph is that collective calls specified with *progress level: none* are additionally prohibited from synchronizing between processes. However, such calls still require the proper collective ordering of matching calls across all participating processes.

4   `UPC++` provides several collective communication operations over teams, described below.

12.1 Common Requirements

¹ For an execution of a `UPC++` program to be valid, the collective operations invoked by the program must obey the following ordering constraints:

² • For a collective operation C over a team T, let $Participants(C)$ denote the set of processes that are members of T.

³ • For a process $P \in Participants(C_1) \cap Participants(C_2)$, let $Precedes_P(C_1, C_2)$ be true if and only if $C_1 \neq C_2$ and C_1 is initiated before C_2 on P.

⁴ • Let *Collectives* be the set of collective operations invoked during execution of the program. The collectives must satisfy the following property:

$$\forall C_1, C_2 \in Collectives. \ \forall P, Q \in Participants(C_1) \cap Participants(C_2).$$
$$Precedes_P(C_1, C_2) = Precedes_Q(C_1, C_2) \tag{12.1}$$

⁵ The constraints above formalize the notion that any two processes that both participate in a pair of collectives must agree on their ordering.

⁶ For any collective operation C, it is an error if the completion of the operation (return from synchronous collectives, operation-completion notifications for asynchronous collectives) on at least one participant has a *happens-before* relationship with the initiation of operation C on another participant.

⁷ When invoking a collective operation, the master persona of the process must appear in the persona stack of the calling thread (§10.5.1). In other words, the expression `master_persona().active_with_caller()` must be true for the calling thread. This property is initially true for the thread that invokes `upcxx::init`, meaning this requirement is trivially satisfied for processes that only ever invoke UPC++ from one thread (provided it never invokes `liberate_master_persona()`).

12.2 API Reference

¹
```
enum class entry_barrier {
  none,
  internal,
  user
};
```

2   Constants used with some `UPC++` operations to specify the entry barrier to be used by the operation:

3   • `none`: the operation has no entry barrier

4   • `internal`: the operation should perform a barrier at entry that makes only internal-level progress

5   • `user`: the operation should perform a barrier at entry that makes user-level progress

6
```
void barrier(const team &team = world());
```

7   *This function is collective (§12.1) over the given team, the team must be active, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

8   Performs a barrier operation over the given team. The call will not return until all processes in the team have entered the call. There is no implied relationship between this call and other in-flight operations.

9   *C++ memory ordering:* With respect to all threads participating in this collective, all evaluations which are *sequenced-before* their respective thread's invocation of this call will have a *happens-before* relationship with all evaluations sequenced after the call.

10   *UPC++ progress level:* `user`

11
```
template<typename Cx=/*...*/>
RType barrier_async(const team &team = world(),
                    Cx &&completions=operation_cx::as_future());
```

12   *This function is collective (§12.1) over the given team, the team must be active, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

13   Initiates an asynchronous barrier operation over the given team. The call will return without waiting for other processes to make the call. Operation completion will be signaled after all other processes in the team have entered the call.

Completions:

14
- *Operation*: Indicates completion of the collective from the viewpoint of the caller, implying that all processes in the given team have entered the collective.

15   *C++ memory ordering:* With respect to all threads participating in this collective, all evaluations which are *sequenced-before* their respective thread's invocation of this call will have a *happens-before* relationship with all evaluations sequenced after the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment).

16   *UPC++ progress level:* `internal`

17
```
constexpr /*see below*/ op_fast_add;
constexpr /*see below*/ op_fast_mul;
constexpr /*see below*/ op_fast_min;
constexpr /*see below*/ op_fast_max;
constexpr /*see below*/ op_fast_bit_and;
constexpr /*see below*/ op_fast_bit_or;
constexpr /*see below*/ op_fast_bit_xor;
```

18   Instances of function-object types that have the following overloaded function-call operator:

19
```
   T operator ()(T a, T b) const;
```

20   The function-object types meet the requirements for the `BinaryOp` template parameter to `reduce_one` and `reduce_all` (e.g. they are referentially transparent and concurrently invokable).

21   For `op_fast_add`, `op_fast_mul`, `op_fast_min`, and `op_fast_max`, the allowed types for `T` are those for which `std::is_arithmetic<T>::value` is true. For `op_fast_bit_and`, `op_fast_bit_or`, and `op_fast_bit_xor`, the allowed types for `T` are those for which `std::is_integral<T>::value` is true.

22   The operation performed by the function-call operator is, respectively: binary `+`, binary `*`, `std::min`, `std::max`, binary `&`, `|`, and `^`. If `T` is `bool`, then `op_fast_add` and `op_fast_max` perform the same operation as `op_fast_bit_or`, and `op_fast_mul` and `op_fast_min` perform the same operation as `op_fast_bit_and`.

```
23  template<typename T, typename BinaryOp, typename Cx=/*...*/>
    RType reduce_one(T value, BinaryOp &&op, intrank_t root,
                     const team &team = world(),
                     Cx &&completions=operation_cx::as_future());
    template<typename T, typename BinaryOp, typename Cx=/*...*/>
    RType reduce_all(T value, BinaryOp &&op,
                     const team &team = world(),
                     Cx &&completions=operation_cx::as_future());
    template<typename T, typename BinaryOp, typename Cx=/*...*/>
    RType reduce_one(const T *src, T *dst, size_t count,
                     BinaryOp &&op, intrank_t root,
                     const team &team = world(),
                     Cx &&completions=operation_cx::as_future());
    template<typename T, typename BinaryOp, typename Cx=/*...*/>
    RType reduce_all(const T *src, T *dst, size_t count,
                     BinaryOp &&op,
                     const team &team = world(),
                     Cx &&completions=operation_cx::as_future());
```

24   *This function is collective (§12.1) over the given team, the team must be active,
        and the master persona (§10.5.1) must appear in the persona stack of the calling
        thread.*

25   *Precondition:* `T` must be TriviallySerializable. `BinaryOp` must be a function-
        object type representing an associative and commutative mathematical oper-
        ation taking two values of type `T` and returning a value implicitly convertible
        to `T`. `BinaryOp` must be referentially transparent and concurrently invokable.
        `BinaryOp` may not invoke any UPC++ routine with a progress level other than
        `none`. In the first and third variants, `root` must be single-valued and a valid
        rank in `team`. In the third variant, `src` and `dst` on the process whose rank
        is `root` in the team may be equal but must not otherwise overlap, and `count`
        must be single-valued across all participants. In the fourth variant, `src` and
        `dst` may be equal but must not otherwise overlap, and `src == dst` and `count`
        must both be single-valued.

26   Performs a reduction operation over the processes in the given team.

27   If the team contains only a single process, then the resulting operation com-
        pletion will produce `value` in the first two variants. In the latter two variants,
        the contents of `src` will be copied to `dst` if `src != dst`.

28   If the team contains more than one process, initiates an asynchronous reduction
        over the values provided by each process. The reduction is performed in some

non-deterministic order by applying `op` to combine values and intermediate results. In the second and fourth variants, the order in which `op` is applied may differ between processes, so the results may differ if `op` is not fully associative and commutative (as with floating-point arithmetic on some operands). In the third and fourth variants, the contents of `src` are combined element-wise across the processes in the team, with the results placed in `dst`.

29   In the first variant, the process whose rank is `root` in `team` receives the result of the reduction as part of operation completion, while the remaining processes receive an undefined value.

30   In the second variant, each process receives the result of the reduction as part of operation completion.

31   In the third variant, operation completion signifies the results have been stored in `dst` on the process whose rank is `root` in `team` and that `src` is no longer in use by the reduction. On the remaining processes, the argument `dst` is ignored, and operation completion signifies that `src` is no longer in use by the reduction.

32   In the fourth variant, operation completion on each process signifies that the results have been stored in `dst` on that process and that `src` is no longer in use by the reduction.

33   *Advice to users:* If `op` is one of `op_fast_*` and `T` is one of the allowed types for `op`, implementations may offload the reduction operations to NIC hardware.

    Completions:

34   • *Operation*: Indicates completion of the collective from the viewpoint of the caller, implying that the results of the reduction are available to this process as described above. In the third and fourth variants, also signifies that the `src` buffer may be modified. In the first two variants, this completion produces a value of type `T`. In the latter two variants, this completion does not produce a value.

35   *C++ memory ordering:* With respect to all threads participating in this collective, all evaluations which are *sequenced-before* any thread's invocation of this call will have a *happens-before* relationship with all evaluations sequenced after the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment) on the threads that receive the results of the collective (on the root process in the first and third variants; on all participating processes in the second and fourth variants).

36   *UPC++ progress level:* `internal`

```
37  template<typename T, typename Cx=/*...*/>
    RType broadcast(T value, intrank_t root,
                    const team &team = world(),
                    Cx &&completions=operation_cx::as_future());

    template<typename T, typename Cx=/*...*/>
    RType broadcast(T *buffer, size_t count, intrank_t root,
                    const team &team = world(),
                    Cx &&completions=operation_cx::as_future());
```

38   *This function is collective (§12.1) over the given team, the team must be active,
      and the master persona (§10.5.1) must appear in the persona stack of the calling
      thread.*

39   *Precondition:* `root` must be single-valued and a valid rank in `team`. In the
      second variant, `count` must be single-valued. The type `T` must be TriviallySe-
      rializable.

40   Initiates an asynchronous broadcast (one-to-all) operation, with rank `root` in
      `team` acting as the producer of the broadcast. In the first variant, `value` will
      be asynchronously sent to all processes in the team, encapsulated in operation
      completion, which will be signaled upon receipt of the value. In the second
      variant, the objects in `[buffer,buffer+count)` of rank `root` in `team` are sent
      to the addresses `[buffer,buffer+count)` provided by the receiving processes.
      Operation completion signals completion of the operation with respect to the
      calling process. For the root, this indicates its buffer is available for reuse, and
      for a receiver, it indicates that the data have been received in its buffer.

      Completions:

41   • *Operation*: In the first variant, indicates that the value provided by the
            root is available at the caller, and this completion produces a value of type
            `T`. In the second variant, indicates completion of the collective from the
            viewpoint of the caller as described above, and this completion does not
            produce a value.

42   *C++ memory ordering:* With respect to all threads participating in this col-
      lective, all evaluations which are *sequenced-before* the producing thread's invo-
      cation of this call will have a *happens-before* relationship with all evaluations
      sequenced after the operation-completion notification actions (future readying,
      promise fulfillment, or persona LPC enlistment).

43   *UPC++ progress level:* `internal`

Chapter 13

Atomics

13.1 Overview

1 `UPC++` supports atomic operations on shared memory locations. Atomicity entails that a read-modify-write sequence on a memory location will happen without interference or interleaving with other concurrently executing atomic operations. Atomicity is not guaranteed if a memory location is concurrently targeted by both atomic and non-atomic operations. The order in which concurrent atomics update the same memory is not guaranteed, not even for successively issued operations by a single process. Ordering of atomics with respect to other asynchronous operations is also not guaranteed. The only means to ensure such ordering is by waiting for one operation to complete before initiating its successor. Note that `UPC++` atomics do not interoperate with `std::atomic`.

2 At this time, it is unclear how `UPC++` will support mixing of atomic and non-atomic accesses to the same memory location. Until this is resolved, users must assume that for the duration of the program, once a memory location is accessed via a `UPC++` atomic, only further atomic operations to that location will have meaningful results (note that even global barrier synchronization does not grant an exception to this rule). This unfortunately implies that deallocation of such memory is unsafe, as that would allow the memory to be reallocated to a context unaware of its constrained condition.

3 All atomic operations are associated with an *atomic domain.* An atomic domain is defined for an integer or floating-point type and a set of operations. Currently, the allowed types are `float`, `double`, and any signed or unsigned integral type with a 32-bit or 64-bit representation. The list of operations is detailed in the API section below. A process's

representative of an atomic domain is an instance of an `atomic_domain` class, and the operations are defined as methods on that class.

4 An atomic domain is created collectively over a team. The result is a semantic binding of `atomic_domain` objects as a *collective object*. We use the phrase *atomic domain* to refer to this semantic binding. An atomic domain must be destroyed by the processes in the team collectively calling the `destroy()` member function, which releases the resources associated with the domain.

5 The use of atomic domains permits selection (at construction) of the most efficient available implementation which can provide correct results for the given set of operations on the given data type. This is important because the best possible implementation of a operation "X" may not be compatible with operation "Y". So, this best "X" can only be used when it is known that "Y" will not be used. This issue arises because a NIC may offload "X" (but not "Y") and use of a CPU-based implementation of "Y" would not be coherent with the NIC performing a concurrent "X" operation.

6 Similar to a mutex, an atomic domain exists independent of the data it applies to. User code is responsible for ensuring that data accessed via a given atomic domain is only accessed via that domain, never via a different domain or without use of a domain.

7 Users may create as many domains as needed to describe their uses of atomic operations, so long as there is at most one domain per atomic datum. If distinct data of the same type are accessed using differing sets of operations, then creation of distinct domains for each operation set is recommended to achieve the best performance on each set.

8 For example, to use atomic fetch-and-add, load and store operations on an `int64_t`, a user must first define a domain as follows:

9
```
atomic_domain < int64_t > ad_i64 ({ atomic_op :: load ,
                                   atomic_op :: store ,
                                   atomic_op :: fetch_add });
```

10 Each atomic operation works on a *global pointer* to the type given when the domain was constructed. The target memory must have affinity to a member of the domain's team.

11 All atomic operations are non-blocking and provide an operation-completion event to indicate completion of the atomic. By default, all operations return futures. So, for example, this is the way to call an atomic operation for the previous example's domain:

```
1   global_ptr < int64_t > x = new_ < int64_t >(0);
2   future < int64_t > f = ad_i64 . fetch_add (x, 2,
3                                       std :: memory_order_relaxed );
4   int64_t res = f . wait ();
```

¹² `UPC++` also provides overloads of fetching atomic operations that write the resulting value into a memory location[1] rather than encapsulating it into a future. This avoids some of the overheads associated with non-empty future or promise completions. The following is an example:

```
1   global_ptr<int64_t> x = new_<int64_t>(0);
2   int64_t res;
3   future<> f = ad_i64.fetch_add(x, 2, &res,
4                                 std::memory_order_relaxed);
5   f.wait();
```

¹³ Atomic domains enable a user to select a subset of operations that are supported in hardware on a given platform, and hence more performant.

13.2   Deviations from IEEE 754

¹ `UPC++` atomics on `float` and `double` are permitted to deviate from the IEEE 754 standard [1], even where `float` and `double` otherwise conform to the standard in the underlying C++ implementation. For example, a `UPC++` atomic may perform a `compare_exchange` operation on floating-point values as if they were integers of the same width, and it may compare floating-point values as if they were sign-and-magnitude-representation integers of the same width. This can lead to non-conforming behavior with respect to NaNs and negative zero.

13.3   API Reference

```
1 enum class atomic_op : /* integral type */ {
    /* Syntax below is only for clarity of presentation:
       All enum values are unspecified */
    load = ..., store,
    compare_exchange,
    add, fetch_add,
    sub, fetch_sub,
    mul, fetch_mul,
    min, fetch_min,
    max, fetch_max,
    bit_and, fetch_bit_and,
```

---

[1]The write into that memory location (`&res` in the example here) is **not** guaranteed to be atomic.

```
   bit_or, fetch_bit_or,
   bit_xor, fetch_bit_xor,
   inc, fetch_inc,
   dec, fetch_dec
};
```

2 ```
template<typename T>
class atomic_domain final;
```

3 *C++ Concepts:* DefaultConstructible, MoveConstructible, MoveAssignable, Destructible

4 ```
template<typename T>
bool atomic_domain<T>::is_active() const;
```

5  Returns whether or not this atomic domain is active.  An atomic domain is active if both the following hold:

6      • It was created via the non-default constructor, or was the target of move construction or move assignment from an active atomic domain.

7      • It has not subsequently been passed as an argument to the move constructor or move assignment operator, nor had its state destroyed by a call to `atomic_domain<T>::destroy()`.

8  ```
template<typename T>
atomic_domain<T>::atomic_domain(std::vector<atomic_op> const &ops,
                                const team &team = world());
```

9 *This function is collective (§12.1) over the given team, the team must be active, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

10 *Precondition:* `T` must be one of the approved atomic types: `float`, `double`, or any signed or unsigned integral type with a 32-bit or 64-bit representation. `T` must be a permitted type for each of the operations in `ops`. The set of operations specified by `ops` must be single-valued (Ch. 12).

11 Constructs an active atomic domain for type `T`, with supported operations `ops`. This instance acts as the calling process's representative in the resulting atomic domain.

12 *UPC++ progress level:* `user`

13
```
template<typename T>
atomic_domain<T>::atomic_domain();
```

14   Constructs an inactive atomic domain.

15   *This function may be called when* `UPC++` *is in the uninitialized state.*

16
```
template<typename T>
atomic_domain<T>::atomic_domain(atomic_domain &&other);
```

17   *Precondition:* The master persona (§10.5.1) must appear in the persona stack of the calling thread.

18   Makes this instance the calling process's representative of the atomic domain associated with `other`, transferring all state from `other`. Deactivates `other`.

19
```
template<typename T>
atomic_domain<T>& atomic_domain<T>::operator=(
                                  atomic_domain &&other);
```

20   *Precondition:* `!this->is_active()`. The master persona (§10.5.1) must appear in the persona stack of the calling thread.

21   Makes this instance the calling process's representative of the atomic domain associated with `other`, transferring all state from `other`. Deactivates `other`.

22
```
template<typename T>
void atomic_domain<T>::destroy(entry_barrier lev =
                                  entry_barrier::user);
```

23   *This function is collective (§12.1) over the team associated with this atomic domain, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

24   *Precondition:* `this->is_active()`. This instance must be the process's representative of the atomic domain. The team associated with this domain must not have been deactivated since the construction of this domain. `lev` must be single-valued (Ch. 12). After the entry barrier (§12.2) specified by `lev` completes, or upon entry if `lev == entry_barrier::none`, all operations on this atomic domain must have signaled operation completion.

25   Destroys the calling process's state associated with the atomic domain. Deactivates this `atomic_domain`.

26   *C++ memory ordering:* If `lev != entry_barrier::none`, with respect to all
       threads participating in this collective, all evaluations which are *sequenced-*
       before their respective thread's invocation of this call will have a *happens-before*
       relationship with all evaluations sequenced after the call.

27   *UPC++ progress level:* `user` if `lev == entry_barrier::user`,
                             `internal` otherwise

28
```
template < typename T >
atomic_domain <T >:: ~ atomic_domain ();
```

29   *Precondition:* Either `UPC++` must have been uninitialized since this domain's
       creation, or this must be an inactive atomic domain.

30   Destructs this `atomic_domain` object.

31   *Advice to users:* `atomic_domain<T>::destroy()` should be used to deactivate
       an active atomic domain before destruction.

32   *This function may be called when* `UPC++` *is in the uninitialized state.*

33
```
template < typename T > template<typename Cx=/*...*/>
RType atomic_domain <T >:: load (
    global_ptr < const T > p, std :: memory_order  order ,
    Cx &&completions=operation_cx::as_future()) const ;
template < typename T > template<typename Cx=/*...*/>
RType atomic_domain <T >:: load (
    global_ptr < const T > p, T *dst , std :: memory_order  order ,
    Cx &&completions=operation_cx::as_future()) const ;
```

34   *Precondition:* `this->is_active()`. `T` must be the only type used by any atomic
       referencing any part of `p`'s target memory for the entire lifetime of `UPC++`. `order`
       must be `std::memory_order_relaxed` or `std::memory_order_acquire`. The
       `atomic_op::load` operation must have been included in the `ops` used to con-
       struct this `atomic_domain`. The target of `p` must have affinity to a member
       of the team associated with this domain, and that team must not have been
       deactivated since the construction of this domain.

35   Initiates an atomic read of the object at location `p`. In the first variant, the
       value read is produced as part of operation completion. In the second variant,
       the value read is written (non-atomically) to `*dst`.

Completions:

36 • *Operation*: Indicates completion of all aspects of the operation: the remote atomic read and transfer of the result are complete. In the first variant, this completion produces a value of type `T`. In the second variant, this completion does not produce a value.

37 *C++ memory ordering:* If `order` is `std::memory_order_acquire` then the read performed will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment).

38 *UPC++ progress level:* `internal`

39 ```
template<typename T> template<typename Cx=/*...*/>
RType atomic_domain<T>::store(
    global_ptr<T> p, T val, std::memory_order order,
    Cx &&completions=operation_cx::as_future()) const;
```

40 *Precondition:* `this->is_active()`. `T` must be the only type used by any atomic referencing any part of `p`'s target memory for the entire lifetime of `UPC++`. `order` must be `std::memory_order_relaxed` or `std::memory_order_release`. The `atomic_op::store` operation must have been included in the `ops` used to construct this `atomic_domain`. The target of `p` must have affinity to a member of the team associated with this domain, and that team must not have been deactivated since the construction of this domain.

41 Initiates an atomic write of `val` to the location `p`. Completion of the write is indicated by operation completion.

*Completions:*

42 • *Operation*: Indicates completion of all aspects of the operation: the transfer of the value and remote atomic write are complete.

43 *C++ memory ordering:* If `order` is `std::memory_order_release` then all evaluations *sequenced-before* this call will have a *happens-before* relationship with the write performed. The write performed will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment).

44 *UPC++ progress level:* `internal`

```
45  template <typename T> template<typename Cx=/*...*/>
    RType atomic_domain <T>::compare_exchange(
        global_ptr<T> p, T val1 , T val2 , std::memory_order order ,
        Cx &&completions=operation_cx::as_future()) const;
    template <typename T> template<typename Cx=/*...*/>
    RType atomic_domain <T>::compare_exchange(
        global_ptr<T> p, T val1 , T val2 , T *dst ,
        std::memory_order  order ,
        Cx &&completions=operation_cx::as_future()) const;
```

46   *Precondition:* `this->is_active()`. T must be the only type used by any atomic referencing any part of `p`'s target memory for the entire lifetime of UPC++. `order` must be `std::memory_order_relaxed`, `std::memory_order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_rel`. The `ops` used to construct this `atomic_domain` must have included the `atomic_op::compare_exchange` operation. The target of `p` must have affinity to a member of the team associated with this domain, and that team must not have been deactivated since the construction of this domain.

47   Initiates the atomic read-modify-write operation consisting of: reading the value of the object located at `p`, and if it is equal to `val1`, writing `val2` back. In the first variant, the value produced by operation completion is the one initially read. In the second variant, the value initially read is written (non-atomically) to `*dst`.

      Completions:

48   • *Operation*: Indicates completion of all aspects of the operation: the transfer of the given value to the recipient, remote atomic update, and transfer of the old value to the initiator are complete. In the first variant, this completion produces a value of type `T`. In the second variant, this completion does not produce a value.

49   *C++ memory ordering:* If `order` is either `std::memory_order_release` or `std::memory_order_acq_rel` then all evaluations *sequenced-before* this call will have a *happens-before* relationship with the atomic action. If `order` is `std::memory_order_acquire` or `std::memory_order_acq_rel` then the atomic action will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or LPC enlistment).

50   *UPC++ progress level:* `internal`

```
51   template<typename T> template<typename Cx=/*...*/>
     RType atomic_domain<T>::binary_key(
         global_ptr<T> p, T val, std::memory_order order,
         Cx &&completions=operation_cx::as_future()) const;
     template<typename T> template<typename Cx=/*...*/>
     RType atomic_domain<T>::fetch_binary_key(
         global_ptr<T> p, T val, std::memory_order order,
         Cx &&completions=operation_cx::as_future()) const;
     template<typename T> template<typename Cx=/*...*/>
     RType atomic_domain<T>::fetch_binary_key(
         global_ptr<T> p, T val, T *dst, std::memory_order order,
         Cx &&completions=operation_cx::as_future()) const;
```

52   *Precondition:* `this->is_active()`. `T` must be the only type used by any ato-
        mic referencing any part of `p`'s target memory for the entire lifetime of `UPC++`,
        and it must be one of the permitted types for the operation. `order` must be
        `std::memory_order_relaxed`, `std::memory_order_acquire`, `std::memory_`
        `order_release`, or `std::memory_order_acq_rel`. The `atomic_op::`*op* oper-
        ation must have been included in the `ops` used to construct this `atomic_domain`,
        where *op* is the following for each variant, respectively: *binary_key*,
        `fetch_`*binary_key*, `fetch_`*binary_key*. The target of `p` must have affinity to a
        member of the team associated with this domain, and that team must not have
        been deactivated since the construction of this domain.

binary_key	**Computation**	Supports `float` and `double`
add	+	yes
sub	−	yes
mul	*	yes
min	std::min	yes
max	std::max	yes
bit_and	&	no
bit_or	|	no
bit_xor	^	no

Table 13.1: Binary atomic arithmetic computations

53   Initiates the atomic read-modify-write operation consisting of: reading the
        value of the object located at `p`, performing the operation corresponding to
        binary_key, and writing the new value back. The operation is performed on
        the value initially read and the `val` argument. In the second variant, the value
        initially read is produced by operation completion. In the third variant, the
        value initially read is written (non-atomically) to `*dst`.

54 The correspondence between *binary_key*, its respective arithmetic operation, and the permitted types is as in Table 13.1. All operations support the integral types.

Completions:

55 - *Operation*: Indicates completion of all aspects of the operation: the transfer of the given value to the recipient and remote atomic update, and transfer of the old value to the initiator in the fetch variants, are complete. This completion does not produce a value in the first or third variant and produces a value of type `T` in the second variant.

56 *C++ memory ordering:* If `order` is either `std::memory_order_release` or `std::memory_order_acq_rel` then all evaluations *sequenced-before* this call will have a *happens-before* relationship with the atomic action. If `order` is `std::memory_order_acquire` or `std::memory_order_acq_rel` then the atomic action will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment).

57 *UPC++ progress level:* `internal`

58
```cpp
template<typename T> template<typename Cx=/*...*/>
RType atomic_domain<T>::unary_key(
    global_ptr<T> p, std::memory_order order,
    Cx &&completions=operation_cx::as_future()) const;
template<typename T> template<typename Cx=/*...*/>
RType atomic_domain<T>::fetch_unary_key(
    global_ptr<T> p, std::memory_order order,
    Cx &&completions=operation_cx::as_future()) const;
template<typename T> template<typename Cx=/*...*/>
RType atomic_domain<T>::fetch_unary_key(
    global_ptr<T> p, T *dst, std::memory_order order,
    Cx &&completions=operation_cx::as_future()) const;
```

59 *Precondition:* `this->is_active()`. `T` must be the only type used by any atomic referencing any part of `p`'s target memory for the entire lifetime of `UPC++`, and it must be one of the permitted types for the operation. `order` must be `std::memory_order_relaxed`, `std::memory_order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_rel`. The `atomic_op::`*op* operation must have been included in the `ops` used to construct this `atomic_domain`, where *op* is the following for each variant, respectively: *unary_key*, `fetch_unary_key`, `fetch_unary_key`. The target of

p must have affinity to a member of the team associated with this domain, and that team must not have been deactivated since the construction of this domain.

unary_key	**Computation**	Supports `float` and `double`
inc	++	yes
dec	--	yes

Table 13.2: Unary atomic arithmetic computations

60   Initiates the atomic read-modify-write operation consisting of: reading the value of the object located at `p`, performing the operation corresponding to *unary_key*, and writing the new value back. The operation is performed on the value initially read. In the second variant, the value initially read is produced by operation completion. In the third variant, the value initially read is written (non-atomically) to `*dst`.

61   The correspondence between *unary_key*, its respective arithmetic operation, and the permitted types is as in Table 13.2. All operations support the integral types.

Completions:

62   • *Operation*: Indicates completion of all aspects of the operation: the transfer of the given value to the recipient and remote atomic update, and transfer of the old value to the initiator in the fetch variants, are complete. This completion does not produce a value in the first or third variant and produces a value of type `T` in the second variant.

63   *C++ memory ordering:* If `order` is either `std::memory_order_release` or `std::memory_order_acq_rel` then all evaluations *sequenced-before* this call will have a *happens-before* relationship with the atomic action. If `order` is `std::memory_order_acquire` or `std::memory_order_acq_rel` then the atomic action will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment).

64   *UPC++ progress level:* `internal`

Chapter 14

Distributed Objects

14.1 Overview

[1] In distributed-memory parallel programming, the concept of a single logical object partitioned over several processes is a useful capability in many contexts: for example, geometric meshes, vectors, matrices, tensors, and associative maps. Since `UPC++` is a communication library, it strives to focus on the mechanisms of communication as opposed to the various programming idioms for managing distribution. However, a basic framework for users to implement their own distributed objects is useful and also enables `UPC++` to provide the user with the following valuable features:

1. Universal distributed object naming: per-object names that can be transmitted to other processes while retaining their meaning.

2. Name-to-this mapping: Mapping between the universal name and the calling process's memory address holding that distributed object's state for the process (the calling process's `this` pointer).

[2] The need for universal distributed object naming stems primarily from RPC-based communication. If one process needs to remotely invoke code on a peer's partition of a distributed object, there needs to be some mutually agreeable identifier for referring to that distributed object. For simplicity, this identifier value should be: identical across all processes so that it may be freely communicated while maintaining its meaning. Moreover, the name should be TriviallyCopyable so that it may be serialized into RPCs efficiently (including with the auto-capture `[=]` lambda syntax), hashable, and comparable so that it works well with standard `C++` containers. `UPC++` provides distributed object names meeting these criteria

as well as the registry for mapping names to and from the calling process's partition of the distributed object.

14.2 Building Distributed Objects

1 Distributed objects can be built with the `upcxx::dist_object<T>` constructor over a specific `team` (defaulting to team `world()`). For all processes in the given team, each process constructs an instance of `dist_object<T>`, supplying a value of type `T` representing this process's instance value. All processes in the team must call this constructor collectively. The call *activates* the distributed object, associating it with a universal name that can be used by any rank in the team to locate the resident instance.

2 Alternatively, a `dist_object<T>` can be constructed non-collectively in an inactive state, and then subsequently activated by a collective call to `activate()` across a given team. In either case, the `dist_object<T>` also becomes associated with the `team` used to collectively activate it.

3 When the `dist_object<T>` is destructed, the `T` value is also destructed. At this point, the name will cease to carry meaning on this process. Thus, the programmer should ensure that no process destructs an active distributed object until all name lookups destined for it complete and all hanging references of the form `T&` or `T*` to the value have expired.

4 The names of `dist_object<T>`'s are encoded by the `dist_id<T>` type. This type is TriviallyCopyable, EqualityComparable, LessThanComparable, hashable, and Trivially-Serializable. It has the members `.here()` and `.when_here()` for retrieving the resident `dist_object<T>` instance associated with the name.

14.3 Ensuring Distributed Existence

1 Activation of a `dist_object<T>` via constructor or `activate()` must be performed in a collective context. However activation does not include synchronization, and thus does not guarantee that after the call all other processes in the team have exited or even reached the activation point. Thus, users are required to guard against the possibility that when an RPC carrying a distributed object's name executes, the recipient process may not yet have an entry for that name in its registry. Possible ways to deal with this include:

    1. Barrier: Before issuing communication containing a `dist_id<T>` for a newly activated distributed object, the relevant team completes a `barrier` to ensure global activation of the `dist_object<T>`.

2. Point to point: Before communicating a `dist_id<T>` with a given process, the initiating process uses some two-party protocol to ensure that the peer has activated the `dist_object<T>`.

3. Asynchronous point-to-point: The user performs no synchronization to ensure remote existence. Instead, an RPC is sent which, upon arrival, waits asynchronously for the peer to activate the distributed object before the dependent RPC callback executes.

2 `UPC++` enables the asynchronous point-to-point approach implicitly when arguments of type *`cq`* `dist_object<T>&` are given to any of the RPC family of functions (see §6.6).

14.4 API Reference

14.4.1 dist_object

```
1  struct inactive_t {
      explicit inactive_t() = default;
   };
   constexpr inactive_t inactive;
```

Tag type to indicate construction of an inactive `dist_object`.

```
2  template<typename T>
   class dist_object;
```

3   *C++ Concepts:* DefaultConstructible, MoveConstructible (when `T` is MoveConstructible and MoveAssignable), MoveAssignable (when `T` is MoveConstructible and MoveAssignable), Destructible

```
4  template<typename T>
   bool dist_object<T>::is_active() const;
```

5   Returns whether or not this `dist_object` is active. A `dist_object` is active if both the following hold:

6   • It was created via a non-default constructor that does not take an `inactive_t` argument, or via the move constructor from an argument that is active, or was the target of move assignment from an argument that is active, or `activate()` has been called on it.

7   • It has not subsequently been passed as an argument to the move constructor or move assignment operator.

8   It is always the case that if `is_active()` is true for a `dist_object`, then `has_value()` is also true for that `dist_object`.

9   *This function may be called when* `UPC++` *is in the uninitialized state.*

```
10  template<typename T>
    bool dist_object<T>::has_value() const;
```

11   Returns whether or not this `dist_object` holds a value of type `T`.   A `dist_object` holds a value if both the following hold:

12   • It was created via a non-default constructor or via the move constructor from an argument that holds a value, or was the target of a move assignment from an argument that holds a value, or `emplace()` has been called on it.

13   • It has not subsequently been the target of a move assignment from an argument that does not hold a value.

14   *This function may be called when* `UPC++` *is in the uninitialized state.*

```
15  template<typename T>
    dist_object<T>::dist_object();
```

16   Constructs an inactive `dist_object` that does not hold a value.

17   *This function may be called when* `UPC++` *is in the uninitialized state.*

```
18  template<typename T>
    template<typename ...Arg>
    dist_object<T>::dist_object(inactive_t, Arg &&...arg);
```

19   Constructs an inactive `dist_object` that holds a value constructed with `T(std::forward<Arg>(arg)...)`.

20   *Exceptions:* May throw any exception thrown by the call `T(std::forward<Arg>(arg)...)`.

21   *This function may be called when* `UPC++` *is in the uninitialized state.*

22
```
template < typename T >
dist_object <T >:: dist_object (T value , const team &team = world ());
```

23 *This function is collective (§12.1) over the given team, the team must be active, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

24 *Precondition:* `T` must be MoveConstructible.

25 Constructs and activates this process's member of the distributed object identified by the collective calling context across `team`. The initial value for this process is given in `value`.

26 Any future previously returned from `dist_id<T>::when_here()` for the corresponding `dist_id<T>` will be readied during a subsequent user-level progress for the master persona.

27 *C++ memory ordering:* Although this is a collective call, it does not provide any synchronization or ordering guarantees between participating threads.

28
```
template < typename T >
template < typename ... Arg >
dist_object <T >:: dist_object ( const team &team , Arg &&... arg );
```

29 *This function is collective (§12.1) over the given team, the team must be active, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

30 Constructs and activates this process's member of the distributed object identified by the collective calling context across `team`. The initial value for this process is constructed with `T(std::forward<Arg>(arg)...)`. The result is undefined if this call throws an exception.

31 Any future previously returned from `dist_id<T>::when_here()` for the corresponding `dist_id<T>` will be readied during a subsequent user-level progress for the master persona.

32 *C++ memory ordering:* Although this is a collective call, it does not provide any synchronization or ordering guarantees between participating threads.

```
33  template < typename T >
    dist_object <T >:: dist_object (
                             std :: initializer_list </* unspecified */>);
```

34   *This function is collective (§12.1) over the **world()** team, the team must be active, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

35   *Precondition:* `T` must be MoveConstructible.

36   Equivalent to `dist_object<T>(T{}, world())`, except that the result is undefined if `T{}` throws an exception. This overload resolves an ambiguity when passing an empty initializer list to the constructor.

37   *C++ memory ordering:* Although this is a collective call, it does not provide any synchronization or ordering guarantees between participating threads.

```
38  template < typename T >
    dist_object <T >:: dist_object ( dist_object <T > && other );
```

39   *Precondition:* The master persona (§10.5.1) must appear in the persona stack of the calling thread. `T` must be MoveConstructible and MoveAssignable.

40   If `other` contains a value, that value is assigned to this object's value via `std::move(*other)`. Note that `other.has_value()` remains true in this case.

41   If `other` does not contain a value, then this object is constructed without a value.

42   If `other.is_active()`, makes this instance the calling process's representative of the distributed object associated with `other`, transferring all state from `other`. Deactivates `other`.

```
43  template < typename T >
    dist_object <T >& dist_object <T >:: operator =( dist_object <T > && other );
```

44   *Precondition:* `!this->is_active()`. The master persona (§10.5.1) must appear in the persona stack of the calling thread. `T` must be MoveConstructible and MoveAssignable.

45   If neither `other` nor this object contains a value, this call has no effect.

46   If `other` contains a value, that value is assigned to this object's value via `std::move(*other)`. Note that `other.has_value()` remains true in this case.

47   If `other` does not contain a value but this object contains a value, `~T()` is invoked to destroy the resident value instance, and `this->has_value()` will subsequently be false.

48   If `other.is_active()`, makes this instance the calling process's representative of the distributed object associated with `other`, transferring all state from `other`. Deactivates `other`.

49
```cpp
template<typename T>
dist_object<T>::~dist_object();
```

50   *Precondition:* If `this->is_active()`, either the master persona (§10.5.1) must appear in the persona stack of the calling thread, or UPC++ must have been uninitialized since the `dist_object<T>` activation.

51   If `this->has_value()`, `~T()` is invoked to destroy the resident value instance.

52   If `this->is_active()`, then this call will destroy the calling process's member of the distributed object, and further lookups on this process using the `dist_id<T>` corresponding to this distributed object will have undefined behavior. If this instance has been deactivated by a move, then this call will not affect the distributed object.

53   *This function may be called when `UPC++` is in the uninitialized state.*

54
```cpp
template<typename T>
template<typename ...Arg>
dist_object<T>::emplace(Arg &&...arg);
```

55   Constructs the contained value with `T(std::forward<Arg>(arg)...)`. If this object already contains a value, `~T()` is first invoked to destroy the previously resident value.

56   If `!this->is_active()` and the call `T(std::forward<Arg>(arg)...)` throws an exception, `this->has_value()` will subsequently be false (any previously resident value will have been destroyed). The exception is propagated out from this call.

57   If `this->is_active()` and the call `T(std::forward<Arg>(arg)...)` throws an exception, the result is undefined.

58   *Exceptions:* If `!this->is_active()`, may throw any exception thrown by the call `T(std::forward<Arg>(arg)...)`.

59   *This function may be called when `UPC++` is in the uninitialized state.*

```
60  template < typename T >
    dist_object <T >:: activate ( const team & team );
```

61   *This function is collective (§12.1) over the given team, the team must be active, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

62   *Precondition:* `!this->is_active() && this->has_value().`

63   Activates this process's member of the distributed object identified by the collective calling context across `team`, and `this->is_active()` subsequently returns true.

64   Any future previously returned from `dist_id<T>::when_here()` for the corresponding `dist_id<T>` will be readied during a subsequent user-level progress for the master persona.

65   *C++ memory ordering:* Although this is a collective call, it does not provide any synchronization or ordering guarantees between participating threads.

```
66  template < typename T >
    dist_id <T > dist_object <T >:: id () const ;
```

67   *Precondition:* `this->is_active().`

68   Returns the universal name that uniquely identifies this distributed object.

```
69  template < typename T >
    team & dist_object <T >:: team ();
    template < typename T >
    const team & dist_object <T >:: team () const ;
```

70   *Precondition:* `this->is_active().` The `team` associated with this distributed object must not have been deactivated since the activation of this distributed object.

71   Retrieves a reference to the `team` instance associated with this distributed object.

```
72  template < typename T >
    T* dist_object <T >:: operator ->() const ;
```

73   *Precondition:* `this->has_value().`

74   Access to the value contained within this object.

75   *This function may be called when* `UPC++` *is in the uninitialized state.*

76
```
template < typename T>
T& dist_object <T>:: operator *() const;
```

77   *Precondition:* `this->has_value()`.

78   Access to the value contained within this object.

79   *This function may be called when* `UPC++` *is in the uninitialized state.*

80
```
template < typename T>
future < deserialized_type_t <T>>
    dist_object <T>:: fetch ( intrank_t rank ) const ;
```

   Preconditions:

81   • `this->is_active()`.

82   • `rank` must be a valid rank in the team associated with this distributed
           object.

83   • `T` must be Serializable, but not `view<U, IterType>`.

84   • `deserialized_type_t<T>` must be MoveConstructible.

85   • `rank`'s instance of this distributed object must not have been destroyed
           by the owning process.

86   • The team associated with this distributed object must not have been de-
           activated since the construction of this distributed object.

87   Asynchronously retrieves a copy of the instance of this distributed object asso-
      ciated with the peer index `rank` in this distributed object's team. The result is
      encapsulated in the returned future. This call is equivalent to:

```
rpc ( team (), rank ,
    []( dist_object <T> & obj ) -> const T& {
      return * obj ;
    }, * this )
```

88   Note that `dist_object<T>::fetch()` correctly handles cases where the tar-
      get process has not yet reached activation of this distributed object, as the
      asynchronous fetch operation will automatically be delayed until that occurs.

89   *UPC++ progress level:* `internal`

14.4.2 dist_id

```
1  template<typename T>
   struct dist_id<T>;
```

2   *C++ Concepts:* DefaultConstructible, TriviallyCopyable, StandardLayoutType, EqualityComparable, LessThanComparable, hashable

3   `UPC++` *Concepts:* TriviallySerializable

4   A universal name that uniquely identifies a distributed object.

```
5  template<typename T>
   dist_id<T>::dist_id();
```

6   Initializes this name to be the invalid ID. All default-constructed `dist_id<T>` objects will receive the same, unique invalid identifier. This enables use of `dist_id<T>()` as a placeholder ID for naming the absence of a distributed object.

```
7  template<typename T>
   future<dist_object<T>&> dist_id<T>::when_here() const;
```

8   *Precondition:* This name must not be the invalid ID. The master persona (§10.5.1) must appear in the persona stack of the calling thread.

9   Retrieves a future representing when the calling process activates the `dist_object<T>` corresponding to this name. The returned future will be readied during a subsequent user-level progress of the master persona after the corresponding distributed object is activated on the calling process. If the corresponding distributed object is already active when this is called, may return a ready future.

10 If the calling process never activates or has already destroyed the distributed object corresponding to this name, the returned future will never be readied.

```
11  template<typename T>
    dist_object<T>& dist_id<T>::here() const;
```

12   *Precondition:* This name must be a valid ID for the calling process. The `dist_object<T>` instance owned by the calling process that is associated with this name must have been previously activated but not yet destroyed. The master persona (§10.5.1) must appear in the persona stack of the calling thread.

13   Retrieves a reference to the calling process's `dist_object<T>` instance associated with this name.

```
14  template<typename T>
    std::ostream& operator<<(std::ostream &os, dist_id<T> id);
```

15   Inserts an unspecified character representation of `id` into the output stream `os`. The textual representation of two objects of type `dist_id<T>` is identical if and only if the two objects compare equal.

Chapter 15

Non-Contiguous One-Sided Communication

15.1 Overview

¹ UPC++ provides functions to perform one-sided communications similar to `rget` and `rput` which are dedicated to handle data stored in non-contiguous locations. These functions are denoted with a suffix added to the type of operation, in increasing order of specialization:

² `{rput,rget}_{irregular,regular,strided}`

³ The most general variant of the API, `{rput,rget}_irregular`, accept iterators over an array or collection of `std::pair` (or `std::tuple`) that contain a local or global pointer to a memory location in the first member while the second member contains the size of the contiguous chunk of memory to be transferred. This variant is capable of expressing non-contiguous RMA of arbitrary shape, but pays the highest overhead in metadata to payload ratio.

⁴ The next set of functions, `{rput,rget}_regular`, operates over contiguous elements of identical size on each side of the transfer, and only requires the caller to provide an array or collection of base pointers to each element.

⁵ Finally, the most specialized set of functions, `{rput,rget}_strided`, provide an interface for expressing translational and transposing copies between arbitrary rectangular sections of densely stored N-dimensional arrays. This specialized pattern requires the least metadata, which is constant in size for a given dimensionality. An example of such a transfer is depicted in Figure 15.1.
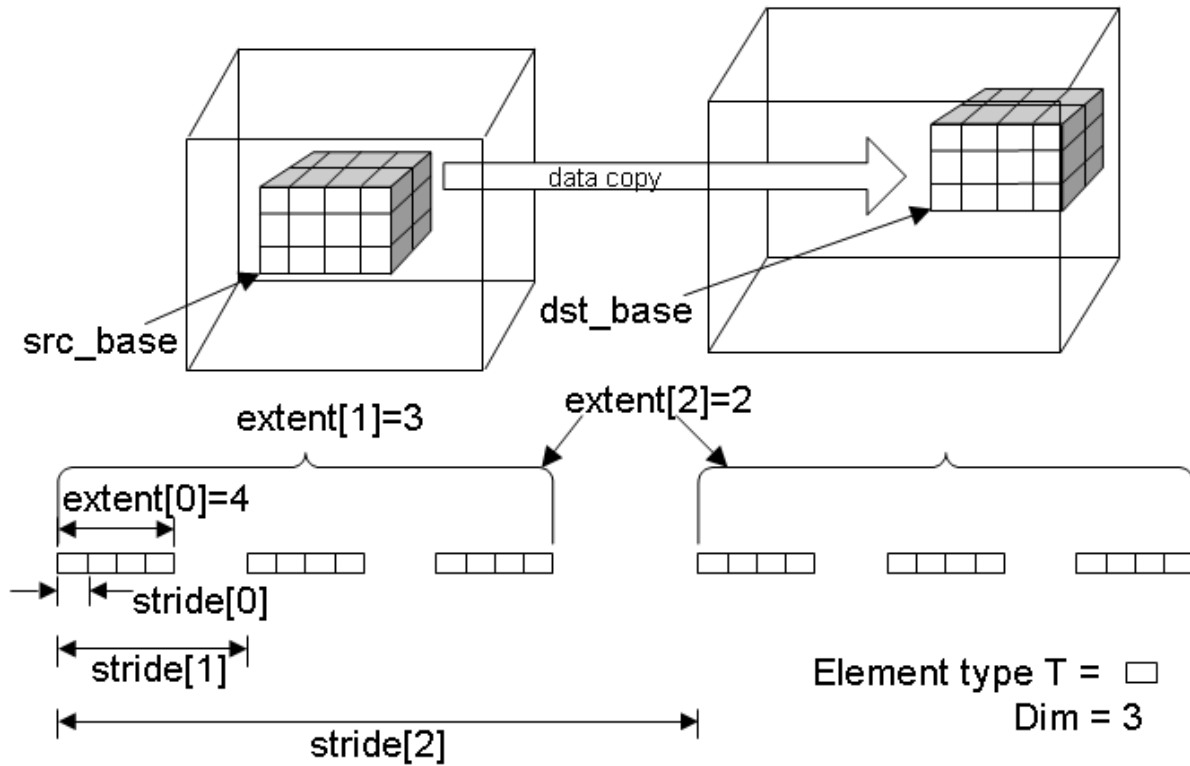
Figure 15.1: Example of a 3-D strided translational copy, with associated metadata

15.2   API Reference

15.2.1   Requirements on Iterators

1 An iterator used with a `UPC++` operation in this section must adhere to the following requirements:

2   • It must satisfy the Iterator and EqualityComparable C++ concepts.

3   • Calling `std::distance` on the iterator must not invalidate it.

15.2.2 Irregular Put

```
1  template<typename SrcIter, typename DestIter, typename Cx=/*...*/>
   RType rput_irregular(
     SrcIter src_runs_begin, SrcIter src_runs_end,
     DestIter dest_runs_begin, DestIter dest_runs_end,
     Cx &&completions=operation_cx::as_future());
```

Preconditions:

2 • `SrcIter` and `DestIter` both satisfy the iterator requirements above.

3 • `std::get<0>(*std::declval<SrcIter>())` has a return type convertible to `T const*`, for some TriviallySerializable type `T`.

4 • `std::get<1>(*std::declval<SrcIter>())` has a return type convertible to `size_t`.

5 • `std::get<0>(*std::declval<DestIter>())` has the return type `global_ptr<T>`, for the same type `T` as with `SrcIter`.

6 • `std::get<1>(*std::declval<DestIter>())` has a return type convertible to `size_t`.

7 • All destination addresses must be `global_ptr<T>`'s referencing memory with affinity to the same process.

8 • The length of the expanded address sequence (the sum over the run lengths) must be the same for the source and destination sequences.

9 • The source and destination addresses must not be null pointers, even if the length of a run is zero.

10 For some type `T`, takes a sequence of source addresses of `T const*` and a sequence of destination addresses of `global_ptr<T>` and does the corresponding puts from each source address to the destination address of the same sequence position.

11 Address sequences are encoded in run-length form as sequences of runs, where each run is a pair consisting of a starting address plus the number of consecutive elements of type `T` beginning at that address.

12 As an example of valid types for individual runs, `SrcIter` could be an iterator over elements of type `std::pair<T const*, size_t>`, and `DestIter` an iterator over `std::pair<global_ptr<T>, size_t>`. Variations replacing

`std::pair` with `std::tuple` or `size_t` with other primitive integral types are also valid.

13   The source sequence iterators must remain valid, and the underlying addresses and source memory contents must not be modified until source completion is signaled. Only after source completion is signaled can the source address sequences and memory be reclaimed by the application.

14   The destination sequence iterators must remain valid until source completion is signaled.

15   The destination memory regions must be completely disjoint and must not overlap with any source memory regions, otherwise behavior is undefined. Source regions are permitted to overlap with each other.

16   Remote-completion operations execute on the master persona of the process associated with the destination (i.e. `dest_runs_begin->where()`), unless the destination sequence is empty (i.e. `dest_runs_begin == dest_runs_end`), in which case they run on the master persona of the initiating process.

Completions:

17   • *Source*: Indicates that the source sequence iterators and underlying memory, as well as the destination sequence iterators, are no longer in use by `UPC++` and may be reclaimed by the user.

18   • *Remote*: Indicates completion of the transfer of all values.

19   • *Operation*: Indicates completion of all aspects of the operation: the transfer and remote stores are complete.

20   *C++ memory ordering:* The reads of the sources will have a *happens-before* relationship with the source-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment). The writes to the destinations will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment) and remote-completion actions (RPC enlistment). For LPC and RPC completions, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of the completion function.

21   *UPC++ progress level:* `internal`

15.2.3 Irregular Get

```
1  template<typename SrcIter, typename DestIter, typename Cx=/*...*/>
   RType rget_irregular(
     SrcIter src_runs_begin, SrcIter src_runs_end,
     DestIter dest_runs_begin, DestIter dest_runs_end,
     Cx &&completions=operation_cx::as_future());
```

Preconditions:

2 - `SrcIter` and `DestIter` both satisfy the iterator requirements above.

3 - `std::get<0>(*std::declval<SrcIter>())` has a type that is convertible to `global_ptr<const T>` for some TriviallySerializable type `T`.

4 - `std::get<1>(*std::declval<SrcIter>())` has a type that is convertible to `size_t`.

5 - `std::get<0>(*std::declval<DestIter>())` has the type `T*`, for the same type `T` as with `SrcIter`.

6 - `std::get<1>(*std::declval<DestIter>())` has a type that is convertible to `size_t`.

7 - All source addresses must be `global_ptr<const T>`'s referencing memory with affinity to the same process.

8 - The length of the expanded address sequence (the sum over the run lengths) must be the same for the source and destination sequences.

9 - The source and destination addresses must not be null pointers, even if the length of a run is zero.

10 For some type `T`, takes a sequence of source addresses of `global_ptr<const T>` and a sequence of destination addresses of `T*` and does the corresponding gets from each source address to the destination address of the same sequence position.

11 Address sequences are encoded in run-length form as sequences of runs, where each run is a pair consisting of a starting address plus the number of consecutive elements of type `T` beginning at that address.

12 As an example of valid types for individual runs, `DestIter` could be an iterator over elements of type `std::pair<T*, size_t>`, and `SrcIter` an iterator over

`std::pair<global_ptr<T>, size_t>`. Variations replacing `std::pair` with `std::tuple` or `size_t` with other primitive integral types are also valid.

13   The source sequence iterators must remain valid, and the underlying addresses and memory contents must not be modified until operation completion is signaled. Only after operation completion is signaled can the address sequences and source memory be reclaimed by the application.

14   The destination sequence iterators must remain valid until operation completion is signaled.

15   The destination memory regions must be completely disjoint and must not overlap with any source memory regions, otherwise behavior is undefined. Source regions are permitted to overlap with each other.

Completions:

16   • *Operation*: Indicates completion of all aspects of the operation: the transfer and local stores are complete.

17   *C++ memory ordering:* The reads of the sources and writes to the destinations will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment). For LPC completions, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of the completion function.

18   *UPC++ progress level:* `internal`

15.2.4 Regular Put

¹
```
template<typename SrcIter, typename DestIter, typename Cx=/*...*/>
RType rput_regular(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  size_t dest_run_length,
  Cx &&completions=operation_cx::as_future());
```

Preconditions:

2
- `SrcIter` and `DestIter` both satisfy the iterator requirements above.

3
- `*std::declval<SrcIter>()` has a type convertible to `T const*`, for some TriviallySerializable type `T`.

4
- `*std::declval<DestIter>())` has the type `global_ptr<T>`, for the same type `T` as with `SrcIter`.

5
- All destination addresses must be `global_ptr<T>`'s referencing memory with affinity to the same process.

6
- The length of the two sequences delimited by (`src_runs_begin`, `src_runs_end`) and (`dest_runs_begin`, `dest_runs_end`) multiplied by `src_run_length` and `dest_run_length`, respectively, must be the same.

7
- The source and destination addresses must not be null pointers, even if `src_run_length` and `dest_run_length` are zero.

8
This call has the same semantics as `rput_irregular` with the exception that, for each sequence, all run lengths are the same and are factored out of the sequences into two extra parameters `src_run_length` and `dest_run_length`, which express the number of consecutive elements of type `T` in units of element count. Thus the iterated elements are no longer pairs, but just pointers.

9
The source sequence iterators must remain valid, and the underlying addresses and source memory contents must not be modified until source completion is signaled. Only after source completion is signaled can the source address sequences and memory be reclaimed by the application.

10
The destination sequence iterators must remain valid until source completion is signaled.

11   Remote-completion operations execute on the master persona of the process associated with the destination (i.e. `dest_runs_begin->where()`), unless the destination sequence is empty (i.e. `dest_runs_begin == dest_runs_end`), in which case they run on the master persona of the initiating process.

Completions:

12
- *Source*: Indicates that the source sequence iterators and underlying memory, as well as the destination sequence iterators, are no longer in use by `UPC++` and may be reclaimed by the user.

13
- *Remote*: Indicates completion of the transfer of all values.

14
- *Operation*: Indicates completion of all aspects of the operation: the transfer and remote stores are complete.

15   *C++ memory ordering:* The reads of the sources will have a *happens-before* relationship with the source-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment). The writes to the destinations will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment) and remote-completion actions (RPC enlistment). For LPC and RPC completions, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of the completion function.

16   *UPC++ progress level:* `internal`

15.2.5   Regular Get

```
template<typename SrcIter, typename DestIter, typename Cx=/*...*/>
RType rget_regular(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  size_t dest_run_length,
  Cx &&completions=operation_cx::as_future());
```

Preconditions:

2
- `SrcIter` and `DestIter` both satisfy the iterator requirements above.

3
- `*std::declval<DestIter>()` has a type convertible to `T*`, for some TriviallySerializable type `T`.

4   • `*std::declval<SrcIter>())` has a type that is convertible to `global_ptr<const T>`, for the same type `T` as with `DestIter`.

5   • All source addresses must be `global_ptr<const T>`'s referencing memory with affinity to the same process.

6   • The length of the two sequences delimited by (`src_runs_begin`, `src_runs_end`) and (`dest_runs_begin`, `dest_runs_end`) multiplied by `src_run_length` and `dest_run_length`, respectively, must be the same.

7   • The source and destination addresses must not be null pointers, even if `src_run_length` and `dest_run_length` are zero.

8   This call has the same semantics as `rget_irregular` with the exception that, for each sequence, all run lengths are the same and are factored out of the sequences into two extra parameters `src_run_length` and `dest_run_length`, which express the number of consecutive elements of type `T` in units of element count. Thus, the iterated elements are no longer pairs, but just pointers.

9   The source sequence iterators must remain valid, and the underlying addresses and memory contents must not be modified until operation completion is signaled. Only after operation completion is signaled can the address sequences and source memory be reclaimed by the application.

10   The destination sequence iterators must remain valid until operation completion is signaled.

Completions:

11   • *Operation*: Indicates completion of all aspects of the operation: the transfer and local stores are complete.

12   *C++ memory ordering:* The reads of the sources and writes to the destinations will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment). For LPC completions, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of the completion function.

13   *UPC++ progress level:* `internal`

15.2.6 Strided Put

```
1  template<size_t Dim, typename T, typename Cx=/*...*/>
   RType rput_strided(
     T const *src_base, ptrdiff_t const *src_strides,
     global_ptr<T> dest_base, ptrdiff_t const *dest_strides,
     size_t const *extents,
     Cx &&completions=operation_cx::as_future());
   template<size_t Dim, typename T, typename Cx=/*...*/>
   RType rput_strided(
     T const *src_base,
     std::array<ptrdiff_t,Dim> const &src_strides,
     global_ptr<T> dest_base,
     std::array<ptrdiff_t,Dim> const &dest_strides,
     std::array<size_t,Dim> const &extents,
     Cx &&completions=operation_cx::as_future());
```

2   *Precondition:* `T` must be a TriviallySerializable type. `src_base` and `dest_base` must not be null pointers, even if the number of bytes to be transferred is zero.

3   If Dim == 0, `src_strides`, `dest_strides`, and `extents` are ignored, and the data movement performed is equivalent to `rput(src_base, dest_base, 1)`.

4   Otherwise, performs the semantic equivalent of many put's of type `T`. Let the *index space* be the set of integer vectors of dimension `Dim` contained in the bounding box with the inclusive lower bound at the all-zero origin, and the exclusive upper bound equal to `extents`. For each index vector `index` in this index space, a put will be executed with addresses computed according to the following pseudo-code, where `dotprod` is the vector dot product and pointer arithmetic is done in units of bytes (not elements of `T`):

5   ```
       src_address = src_base + dotprod(index, src_strides)
       dest_address = dest_base + dotprod(index, dest_strides)
       ```

6   Note this implies the elements of the `src_strides` and `dest_strides` arrays are expressed in units of bytes.

7   The destination memory regions must be completely disjoint and must not overlap with any source memory regions, otherwise behavior is undefined. Source regions are permitted to overlap with each other.

8   The elements of type `T` residing in the source addresses must remain valid and unmodified until source completion is signaled.

9   The contents of the `src_strides`, `dest_strides`, and `extents` arrays are consumed synchronously before the call returns.

10   Remote-completion operations execute on the master persona of the process associated with the destination (i.e. `dest_base.where()`).

     Completions:

11   • *Source*: Indicates that the source memory is no longer in use by `UPC++` and may be reclaimed by the user.

12   • *Remote*: Indicates completion of the transfer of all values.

13   • *Operation*: Indicates completion of all aspects of the operation: the transfer and remote stores are complete.

14   *C++ memory ordering:* The reads of the sources will have a *happens-before* relationship with the source-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment). The writes to the destinations will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment) and remote-completion actions (RPC enlistment). For LPC and RPC completions, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of the completion function.

15   *UPC++ progress level:* `internal`

15.2.7   Strided Get

```
template<size_t Dim, typename T, typename Cx=/*...*/>
RType rget_strided(
  global_ptr<const T> src_base, ptrdiff_t const *src_strides,
  T *dest_base, ptrdiff_t const *dest_strides,
  size_t const *extents,
  Cx &&completions=operation_cx::as_future());
template<size_t Dim, typename T, typename Cx=/*...*/>
RType rget_strided(
  global_ptr<const T> src_base,
  std::array<ptrdiff_t,Dim> const &src_strides,
  T *dest_base,
  std::array<ptrdiff_t,Dim> const &dest_strides,
  std::array<size_t,Dim> const &extents,
  Cx &&completions=operation_cx::as_future());
```

2   *Precondition:* `T` must be a TriviallySerializable type. `src_base` and `dest_base` must not be null pointers, even if the number of bytes to be transferred is zero.

3   If Dim == 0, `src_strides`, `dest_strides`, and `extents` are ignored, and the data movement performed is equivalent to `rget(src_base, dest_base, 1)`.

4   Otherwise, performs the reverse direction of `rput_strided` where now the source memory is remote and the destination is local.

5   The destination memory regions must be completely disjoint and must not overlap with any source memory regions, otherwise behavior is undefined. Source regions are permitted to overlap with each other.

6   The elements of type `T` residing in the source addresses must remain valid and unmodified until operation completion is signaled.

7   The contents of the `src_strides`, `dest_strides`, and `extents` arrays are consumed synchronously before the call returns.

Completions:

8   • *Operation*: Indicates completion of all aspects of the operation: the transfer and local stores are complete.

9   *C++ memory ordering:* The reads of the sources and writes to the destinations will have a *happens-before* relationship with the operation-completion notification actions (future readying, promise fulfillment, or persona LPC enlistment). For LPC completions, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of the completion function.

10   *UPC++ progress level:* `internal`

Chapter 16

Memory Kinds

16.1 Overview

1   The memory kinds interface enables the programmer to identify regions of memory requiring different access methods or having different performance properties, and subsequently rely on the `UPC++` communication services to perform transfers among such regions (both local and remote) in a manner transparent to the programmer. With GPU devices, HBM, scratch-pad memories, NVRAM and various types of storage-class and fabric-attached memory technologies featured in vendors' public road maps, `UPC++` must be prepared to deal efficiently with data transfers among all the memory technologies in any given system.

2   `UPC++` uses *device* objects to represent storage that is distinct from main memory, regardless of whether the storage is directly addressable from the host process. Each kind of memory has its own device type; for example, a CUDA-enabled GPU device is represented by a `cuda_device` object. The device type has a member type-alias template `pointer` that refers to the device's pointer type, as well as a `null_pointer` member-function template that returns a null-pointer value of that type. Each device type is associated with a `memory_kind` constant, and global pointers are parameterized by a `memory_kind` (Ch. 3).

3   Creating active device objects is a collective operation over the `world()` team so that `UPC++` can allocate the resources required to support remote access to device memory. The result is a semantic binding of device objects as a collective object, which we refer to as a *collective device.* A device type also provides a mechanism for constructing inactive device objects, so that processes without a device resource can still participate in the collective device-creation operation. A collective device must be destroyed by collectively calling the

destroy() member function, which releases the resources associated with the collective device.

```
1  cuda_device :: id_type device_id =
2    rank_me () % 2 == 0 ? 0 : cuda_device :: invalid_device_id ;
3  cuda_device device ( device_id ) ; // device 0 on even processes
4  ...
5  device . destroy () ; // collective destroy
```

4 A device object can be associated with a `device_allocator` that manages memory on the device. Only one `device_allocator` may be associated with a particular device object (however multiple device objects may optionally be constructed for the same physical device). The region of memory that a `device_allocator` manages is called a *device segment*. Users can either allocate their own memory segments and pass them to the `device_allocator` constructor, or they can request that the `device_allocator` allocate its own segment. In the latter case, the segment is automatically freed when the `device_allocator` is destroyed.

```
1  size_t seg_size = 4*1024*1024; // 4MB
2  device_allocator < cuda_device > gpu_alloc ( device , seg_size ) ;
3  global_ptr < double , memory_kind :: cuda_device > gpu_array =
4    gpu_alloc . allocate < double >(1024) ;
5  ...
6  gpu_alloc . deallocate ( gpu_array ) ;
```

5 We define the affinity (Ch. 3) of memory allocated by a `device_allocator` to be the host process that owns the allocator and its associated device.

6 The device types defined in this section are available to `UPC++` programs even in `UPC++` installations that are not aware of a particular kind of device; for example, a program may still create `cuda_device` objects. However, there are no valid CUDA device IDs in a non-CUDA-aware installation, so any `cuda_device` object created by that program will be inactive. Feature macros are provided to advertise kinds supported by the installation, for example macro `UPCXX_KIND_CUDA` is defined if and only if the installation is CUDA-aware.

7 There is also a simplified interface for opening a GPU device and constructing a corresponding `device_allocator` to manage the device segment, in a single step:

```
1  // make allocator for a device segment on the default GPU:
2  gpu_heap_allocator gpu_alloc = make_gpu_allocator ( seg_size ) ;
3
4  global_ptr < double , memory_kind :: any > gpu_array =
5    gpu_alloc . allocate < double >(1024) ;
6  ...
```

```
7    gpu_alloc.deallocate(gpu_array);
8
9    gpu_alloc.destroy(); // release device segment and close device
```

The first line in this example creates a `device_allocator` for the default GPU kind defined by the UPC++ implementation (i.e., generally a site-specific setting), automatically choosing an available device of that kind.

8 The `copy` functions transfer data between memory locations of any kind. The source and destination locations may either be local or remote, and they may refer to either host or device memory.

```
1    global_ptr<double> host_array = new_array<double>(1024);
2    global_ptr<double, memory_kind::any> array0 =
3      broadcast(gpu_array, 0).wait();
4    // copy from gpu array on process 0 to host array on this process
5    copy(array0, host_array, 1024).wait();
```

16.2   API Reference

```
1    #define UPCXX_KIND_CUDA 202309L
```

2   A macro definition to an integer literal identifying the version of the CUDA memory-kind feature. CUDA-aware installations of `UPC++` that conform to this specification shall define the value shown above. Non-CUDA-aware installations shall leave the macro undefined. Future versions of this specification may replace the value of this macro with a greater value.

```
3    #define UPCXX_KIND_HIP 202309L
```

4   A macro definition to an integer literal identifying the version of the ROCm/HIP memory-kind feature. HIP-aware installations of `UPC++` that conform to this specification shall define the value shown above. Non-HIP-aware installations shall leave the macro undefined. Future versions of this specification may replace the value of this macro with a greater value.

5 `#define UPCXX_KIND_ZE 202309L`

6   A macro definition to an integer literal identifying the version of the oneAPI Level-Zero (ZE) memory-kind feature. ZE-aware installations of `UPC++` that conform to this specification shall define the value shown above. Non-ZE-aware installations shall leave the macro undefined. Future versions of this specification may replace the value of this macro with a greater value.

7 `class bad_segment_alloc : public std::bad_alloc;`

8   An exception type derived from `std::bad_alloc` that is thrown by some shared segment constructors to indicate failure to allocate resources needed to create the memory segment.

16.2.1   gpu_device

1 `struct gpu_device;`

2   *C++ Concepts:* Destructible

       An abstract base class representing access to a generic GPU device.

3 `memory_kind gpu_device::kind() const;`

4   Returns the `memory_kind` value corresponding to this object. For example, an instance of `cuda_device` would return `memory_kind::cuda_device`.

5 `std::string gpu_device::kind_info() const;`

6   Returns a textual representation summarizing hardware configuration information for devices available to the caller of kind `this->kind()`, in an unspecified human-readable format.

7 `virtual bool gpu_device::is_active() const;`

8   Returns whether or not this `gpu_device` is active. A `gpu_device` is active if both the following hold:

9   • It was created with a valid device ID, or was the target of move construction or move assignment from an active `gpu_device`.

10   • It has not subsequently been passed as an argument to the move constructor or move assignment operator, nor had its state destroyed by a call to `destroy()`.

```
11 virtual void gpu_device::destroy(entry_barrier lev =
                                    entry_barrier::user);
```

12 *This function is collective (§12.1) over the* **world()** *team, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

13 *Precondition:* If this process's representative of the collective device being destroyed is inactive, then this `gpu_device` must be inactive. Otherwise, this instance must be the process's representative of the collective device. `lev` must be single-valued (Ch. 12). After the entry barrier (§12.2) specified by `lev` completes, or upon entry if `lev == entry_barrier::none`, all asynchronous `UPC++` operations on memory associated with this device must have signaled operation completion.

14 Destroys the calling process's state associated with this `gpu_device`, deactivating this device object.

15 Deactivates any `device_allocator` associated with this device object, and any global pointers referencing memory in the associated device segment are invalidated. If the device segment was allocated by the `device_allocator` constructor, additionally frees the associated device segment, without invoking any destructors for objects in the segment. The contents of device memory remain otherwise unchanged.

16 *C++ memory ordering:* If `lev != entry_barrier::none`, with respect to all threads participating in this collective, all evaluations which are *sequenced-before* their respective thread's invocation of this call will have a *happens-before* relationship with all evaluations sequenced after the call.

17 *UPC++ progress level:* user if `lev == entry_barrier::user`, internal otherwise

```
18 virtual gpu_device::~gpu_device();
```

19 *Precondition:* Either `UPC++` must have been uninitialized since the construction of this `gpu_device`, or this must be an inactive `gpu_device`.

20 Destructs this `gpu_device` object. The contents of device memory remain unchanged.

21 *Advice to users:* `gpu_device::destroy()` or `device_allocator::destroy()` should be used to deactivate an active `gpu_device` before destruction.

22 *This function may be called when* `UPC++` *is in the uninitialized state.*

16.2.2 cuda_device, hip_device, ze_device

```
1  class cuda_device final : public gpu_device;
   class hip_device final : public gpu_device;
   class ze_device final : public gpu_device;
```

2   *C++ Concepts:* DefaultConstructible, MoveConstructible, MoveAssignable, Destructible

3   `cuda_device` represents access to a GPU device supporting the CUDA API, such as provided by NVIDIA-branded GPU devices.

4   `hip_device` represents access to a GPU device supporting the ROCm/HIP API, such as provided by AMD-branded GPU devices.

5   `ze_device` represents access to a GPU device supporting the oneAPI Level-Zero API, such as provided by Intel-branded GPU devices.

6   Each is a concrete class derived from the abstract base class `gpu_device`, and inherits the members specified in the previous section for that base class.

7   The remainder of this section specifies semantically analogous members of each concrete class, where *GpuDevice* refers to any of these concrete types.

```
8  static constexpr memory_kind cuda_device::kind =
                   memory_kind::cuda_device;
   static constexpr memory_kind hip_device::kind =
                   memory_kind::hip_device;
   static constexpr memory_kind ze_device::kind =
                   memory_kind::ze_device;
```

9   Constant that has the same value returned by `gpu_device::kind()`.

```
10 class GpuDevice {
     using id_type = int;
     // ...
   };
```

11   Member alias for the type of a valid GPU device ID.

```
12  class GpuDevice {
      template<typename T>
      using pointer = T*;
      // ...
    };
```

13   Member template alias for raw device pointer types on the GPU device.

```
14  template<typename T>
    static constexpr GpuDevice::pointer<T> GpuDevice::null_pointer();
```

15   Returns a representation of a null GPU device pointer.

```
16  template<typename T>
    static constexpr size_t GpuDevice::default_alignment();
```

17   Returns the default alignment of an object of type T on the GPU device.

```
18  static constexpr
    GpuDevice::id_type GpuDevice::invalid_device_id = /* see below */;
    static constexpr
    GpuDevice::id_type GpuDevice::auto_device_id = /* see below */;
```

19   *GpuDevice*::`invalid_device_id` is a constant representing an invalid GPU
      device ID.

20   *GpuDevice*::`auto_device_id` is a distinct invalid GPU device ID that activates
      different behavior when passed to certain functions. This constant shall only be
      passed as a device ID to functions explicitly specified to allow it (see §16.2.5).

```
21  static GpuDevice::id_type GpuDevice::device_n();
```

22   Returns the number of valid GPU device IDs of the given variety available to
      the calling process. Valid device IDs are integers in the range `[0,device_n())`.
      Note that each variety of GPU has an independent device ID space; for example,
      CUDA device ID 0 and HIP device ID 0 may in general denote different physical
      GPU devices.

```
23  static std::string GpuDevice::kind_info();
```

24   Returns a textual representation summarizing hardware configuration informa-
      tion for devices available to the caller of kind *GpuDevice*::`kind`, in an unspec-
      ified human-readable format.

25 `static std::string` *GpuDevice*`::uuid(`*GpuDevice*`::id_type device_id);`

26   *Precondition:* `device_id` must be in the range `[0,device_n())`.

27   Returns a textual representation of a device-specific identifier associated with
        the hardware device named by `device_id`, in an unspecified human-readable
        format.

28   *Advice to users:* GPU Device IDs in `[0,device_n())` enumerate the set of GPU
        devices visible to a given process. The set of valid Device IDs and their corre-
        spondence to hardware devices may differ in unspecified ways across processes.
        The intent of the `uuid()` function is to provide hardware-level identification
        information corresponding to a given Device ID that ideally can be used to
        uniquely identify a given hardware device and allow meaningful comparison
        between processes.

29 *GpuDevice*`::`*GpuDevice*`();`

30   Constructs an inactive *GpuDevice* object.

31   *This function may be called when* `UPC++` *is in the uninitialized state.*

32 *GpuDevice*`::`*GpuDevice*`(`*GpuDevice*`::id_type device_id);`

33   *This function is collective (§12.1) over the* `world()` *team, and the master
        persona (§10.5.1) must appear in the persona stack of the calling thread.*

34   *Precondition:* `device_id` must be *GpuDevice*`::invalid_device_id` or in the
        range `[0,device_n())`.

35   If the device ID is *GpuDevice*`::invalid_device_id`, constructs an inactive
        GpuDevice object. Otherwise, constructs an active *GpuDevice* with the given
        device ID, which acts as the calling process's representative of the resulting
        collective device.

36   The contents of device memory remain unchanged.

37   *UPC++ progress level:* `user`

38 *GpuDevice*`::`*GpuDevice*`(`*GpuDevice* `&&other);`

39   Transfers the state represented by `other` to this *GpuDevice.* Deactivates `other`.

40   The contents of device memory remain unchanged.

41 *GpuDevice* `&` *GpuDevice* `:: operator =(` *GpuDevice* `&& other );`

42   *Precondition:* `!this->is_active().`

43   Transfers the state represented by `other` to this *GpuDevice.* Deactivates `other`.

44   The contents of device memory remain unchanged.

45 *GpuDevice* `:: id_type` *GpuDevice* `:: device_id () const ;`

46   Returns the device ID of this *GpuDevice.* If this is an inactive device, returns *GpuDevice* `:: invalid_device_id.`

16.2.3   heap_allocator

1 `struct heap_allocator ;`

2   *C++ Concepts:* Destructible

3   An abstract base class representing an allocator for a shared memory segment.

4   `heap_allocator` is the base class for `device_allocator<Device>` (see §16.2.4), and the latter inherits all of the following members.

5 `memory_kind heap_allocator :: kind () const ;`

6   Returns the `memory_kind` value corresponding to the shared memory kind associated with this object's dynamic type. For example, an instance of `device_allocator<cuda_device>` would return `memory_kind::cuda_device`.

7 `virtual bool heap_allocator :: is_active () const ;`

8   Returns whether or not this `heap_allocator` is active. A `heap_allocator` is active if both the following hold:

9   • It was created with an active device, or was the target of move construction or move assignment from an active `heap_allocator`.

10   • It has not subsequently been passed as an argument to the move constructor or move assignment operator, nor had its state destroyed by a call to `destroy()`.

```
11  virtual std::int64_t heap_allocator::segment_size() const;
```

12   If `this->is_active()`, requests a snapshot of the total size of the segment controlled by this allocator. Otherwise, returns zero.

13   Implementations are permitted to return unspecified negative values, which indicate the query is unsupported or encountered some other error.

14   A positive return value indicates the current total size, in bytes, of the segment controlled by this allocator. This total size reflects an upper bound on the sum of space consumed by all objects allocated but not yet deallocated using `*this`, space available for servicing subsequent such requests, and unspecified implementation overheads (including but not limited to, padding around allocated objects and objects allocated by the implementation).

```
15  virtual std::int64_t heap_allocator::segment_used() const;
```

16   If `this->is_active()`, requests a snapshot of the occupied size of the segment controlled by this allocator. Otherwise, returns zero.

17   Implementations are permitted to return unspecified negative values, which indicate the query is unsupported or encountered some other error.

18   A positive return value indicates the current occupied size, in bytes, of the segment controlled by this allocator. This occupied size reflects an upper bound on the sum of space consumed by all objects allocated but not yet deallocated using `*this`, and unspecified implementation overheads (including but not limited to, padding around allocated objects and objects allocated by the implementation).

```
19  template<typename T>
    global_ptr<T, memory_kind::any>
        heap_allocator::allocate(size_t n = 1, size_t align = 0);
```

20   *Precondition:* `align` is zero or a valid alignment. Additional preconditions from the `allocate` member function template on the dynamic type of `*this` also apply.

21   This function internally performs a virtual dispatch to the `allocate` member function template of the appropriate derived type.

22   When this object is an instance of `device_allocator<Device>`, this call returns a global pointer obtained as if by calling:

```
              static_cast<device_allocator<Device> *>(this)->
                                        allocate<T>(n,align)
```

and converting the resulting value to `global_ptr<T,memory_kind::any>`. Additionally, when the provided `align` is zero, it is treated as if `Device::default_alignment<T>()` was instead passed for `align`.

23 The preconditions and semantics are otherwise identical to the corresponding `allocate` member function template of the appropriate derived type.

24 ```
template<typename T, memory_kind K>
void heap_allocator::deallocate(global_ptr<T, K> g);
```

25  *Precondition:* `K == this->kind() || K == memory_kind::any`. Additional preconditions from the `deallocate` member function template on the dynamic type of `*this` also apply.

26  This function internally performs a virtual dispatch to the `deallocate` member function template of the appropriate derived type.

27  When this object is an instance of `device_allocator<Device>`, this call is semantically equivalent to:

```
              static_cast<device_allocator<Device> *>(this)->
                                        deallocate<T>(g)
```

28 The preconditions and semantics are otherwise identical to the corresponding `deallocate` member function template of the appropriate derived type.

29 ```
virtual void heap_allocator::destroy(entry_barrier lev =
                          entry_barrier::user);
```

30 Equivalent to calling `device_object.destroy(lev)`, where `device_object` is the device object associated with this allocator. For example, an instance of `device_allocator<cuda_device>` will invoke `cuda_device::destroy()` on the associated `cuda_device` object.

31 The preconditions and semantics are otherwise identical to the corresponding `destroy` member function of the appropriate device type.

32 ```
virtual heap_allocator::~heap_allocator();
```

33  Destructs this `heap_allocator` object by dispatching to the destructor provided by the derived allocator type.

34  *This function may be called when* `UPC++` *is in the uninitialized state.*

### 16.2.4 device_allocator

```
1  template < typename Device >
   class device_allocator final : public heap_allocator;
```

2   *C++ Concepts:* DefaultConstructible, MoveConstructible, MoveAssignable,
     Destructible

3   A class template representing an allocator for a device segment.

4   `device_allocator<Device>` is a concrete type derived from the abstract base
     class `heap_allocator`, and inherits the members specified in the previous sec-
     tion for that base class.

```
5  template < typename Device >
   class device_allocator {
     using device_type = Device;
     // ...
   };
```

6   Member type that is an alias for the template parameter `Device`.

```
7  template < typename Device >
   class device_allocator {
     static constexpr memory_kind kind = Device::kind;
     // ...
   };
```

8   A constant indicating the memory kind managed by this allocator type.

```
9  template < typename Device >
   device_allocator < Device >:: device_allocator ();
```

10   Constructs an inactive `device_allocator` object.

11   *This function may be called when* `UPC++` *is in the uninitialized state.*

12
```
template < typename  Device >
device_allocator < Device >:: device_allocator(
                        Device &device , size_t sz_in_bytes ,
                        typename  Device :: pointer < void > device_memory =
                          Device :: template  null_pointer < void >());
```

13   *This function is collective (§12.1) over the* ***world()*** *team, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

    Preconditions:

14   • Either `device` is inactive, or `device` must not have been previously used to create a `device_allocator`.

15   • If `device` is inactive, the other arguments provided by that caller are ignored. Otherwise, the following preconditions apply.

16   • `sz_in_bytes` must be non-zero.

17   • If `device_memory` is non-null, then it must be a pointer to memory at least `sz_in_bytes` bytes in size that is associated with the given device, and it must not be managed by another `device_allocator`.

18   Although this operation is collective, the arguments need not be single-valued. If `device` is inactive, then that caller constructs an inactive `device_allocator` object. Otherwise, the caller constructs an active `device_allocator` associated with the given device to manage a device segment.

19   If `device` is active and `device_memory` is non-null, then the `device_allocator` constructed by this caller manages the given `device_memory` as its device segment, and this call does not alter the contents of `device_memory`. The provided device memory must remain valid until the `destroy()` of the constructed `device_allocator`.

20   If `device` is active and `device_memory` is null, then this caller attempts to allocate memory of at least size `sz_in_bytes` bytes from the given device to serve as the device segment. The segment is allocated from the associated device in a device-specific manner. Any device-specific properties of the resulting allocation are implementation-defined. If special properties are required, users may supply their own device segment instead using the `device_memory` argument.

21   If segment allocation fails for any callers participating in this collective with an active device and null `device_memory`, then any successful allocations by any caller are released and all callers will throw `upcxx::bad_segment_alloc`.

22   *Exceptions:* May throw `upcxx::bad_segment_alloc`.

23   *UPC++ progress level:* `user`

```
24  template < typename Device >
    device_allocator < Device >:: device_allocator (
                                device_allocator && other );
```

25   Transfers the state represented by `other` to this `device_allocator`. Deactivates `other`.

26   The contents of device memory remain unchanged.

```
27  template < typename Device >
    device_allocator < Device >& device_allocator < Device >:: operator =(
                                device_allocator && other );
```

28   *Precondition:* `!this->is_active()`.

29   Transfers the state represented by `other` to this `device_allocator`. Deactivates `other`.

30   The contents of device memory remain unchanged.

```
31  template < typename Device >
    device_allocator < Device >::~ device_allocator () override ;
```

32   *Precondition:* Either `UPC++` must have been uninitialized since the construction of this `device_allocator`, or this must be an inactive `device_allocator`.

33   Destructs this `device_allocator` object. The contents of device memory are unchanged.

34   *Advice to users:* Either `device_allocator::destroy()` or `Device::destroy()` should be used to deactivate an active `device_allocator` before destruction.

35   *This function may be called when* `UPC++` *is in the uninitialized state.*

```
36  template < typename Device >
    template < typename T >
    global_ptr <T, Device :: kind >
        device_allocator < Device >:: allocate ( size_t n = 1 ,
                size_t align = Device :: default_alignment <T >());
```

37   *Precondition:* `this->is_active()`. `align` is a valid alignment.

38   Allocates enough space for `n` objects of type `T` from the segment managed by
        this allocator, with the memory aligned as specified by `align`. If the allocation
        succeeds, returns a global pointer to the start of the allocated memory, and the
        allocated memory is uninitialized. If the allocation fails, returns a null pointer.

```
39  template < typename Device >
    template < typename T >
    void device_allocator < Device >:: deallocate (
                                        global_ptr <T, Device :: kind > g );
    template < typename Device >
    template < typename T >
    void device_allocator < Device >:: deallocate (
                                        global_ptr <T, memory_kind :: any > g );
```

40   *Precondition:*
        `g.is_null() || (this->is_active() && g.where == rank_me())`.
        `g` must be either a null pointer or a non-deallocated pointer that resulted from
        a call to `allocate<T, align>` on this allocator, for some value of `align`.

41   Deallocates the storage previously allocated by a call to `allocate`. Does noth-
        ing if `g` is a null pointer. Does not invoke the destructor for `T`.

```
42  template < typename Device >
    template < typename T >
    global_ptr <T, Device :: kind >
        device_allocator < Device >:: to_global_ptr (
                                    typename Device :: pointer <T > ptr ) const ;
```

43   *Precondition:* `ptr` is a null pointer, or `this->is_active()` and `ptr` is a valid
        pointer such that the expression `*ptr` on this allocator's device yields a (possi-
        bly uninitialized) object of type `T` that resides within the segment managed by
        this allocator.

44   Converts a raw device pointer to a global pointer.

```
45 template < typename Device >
   template < typename T >
   static typename Device :: pointer <T>
     device_allocator < Device >:: local ( global_ptr <T, Device :: kind > g );
```

46   *Precondition:* `g.is_null() || g.where() == rank_me()`. g must be either a null pointer or a valid pointer to a (possibly uninitialized) object of type `T` that resides within a device segment currently managed by an active `device_allocator<Device>` on the caller's process.

47   Returns the raw device pointer associated with `g`. If `g` is a null pointer, returns `Device::null_pointer<T>()`.

```
48 template < typename Device >
   typename Device :: id_type
       device_allocator < Device >:: device_id () const;
```

49   If `this->is_active()`, returns the ID of the device associated with this allocator. Otherwise, returns `Device::invalid_device_id`.

```
50 template < typename Device >
   template < typename T >
   static typename Device :: id_type
       device_allocator < Device >:: device_id (
                           global_ptr <T, Device :: kind > g );
```

51   *Precondition:* `g.is_null() || g.where() == rank_me()`. g must be either a null pointer or a valid pointer to a (possibly uninitialized) object of type `T` that resides within a device segment currently managed by an active `device_allocator<Device>` on the caller's process.

52   If the pointer is not null, returns the ID of the device where the referenced object resides. If the pointer is null, returns `Device::invalid_device_id`.

16.2.5 Simplified Allocator Management

```
1 using gpu_default_device = /* see below */;
```

2   A type alias for a derived type of `gpu_device` that indicates the default GPU device type.

3   The choice of default GPU device type is implementation-defined.

4 ```
using gpu_heap_allocator = device_allocator < gpu_default_device >;
```

5   A type alias for the default GPU device allocator.

6 ```
template < typename Device = gpu_default_device >
device_allocator < Device > make_gpu_allocator (size_t sz_in_bytes ,
            Device :: id_type device_id = Device :: auto_device_id ,
            void *device_memory = nullptr);
```

7       *This function is collective (§12.1) over the **world()** team, and the master persona (§10.5.1) must appear in the persona stack of the calling thread.*

8       *Precondition:* `Device` is a derived type of `gpu_device`. `device_id` must be `Device::invalid_device_id`, `Device::auto_device_id`, or in the range `[0,Device::device_n())`. If `device_memory` is non-null, then `device_id` must not be `Device::auto_device_id`.

9       Although this operation is collective, the arguments need not be single-valued.

10       If `device_id` is `Device::auto_device_id`, constructs a `Device` object `device_object` as if by calling `Device(auto_id)`, where `auto_id` is an implementation-defined choice of valid device ID, or `Device::invalid_device_id` if no valid device ID is available.

11       Otherwise, constructs a `Device` object `device_object`, as if by calling `Device(device_id)`.

12       The lifetime of `Device` object `device_object` is managed internally.

13       If the resulting `device_object` is inactive, the other arguments provided by that caller are ignored and the call returns an inactive `device_allocator<Device>`.

14       If the resulting `device_object` is active and `device_memory` is null, constructs an active `device_allocator<Device>` to allocate and manage a GPU device segment of size `sz_in_bytes` (which must be non-zero) as if by calling `device_allocator<Device>(device_object, sz_in_bytes)` and returns the result.

15       If the resulting `device_object` is active and `device_memory` is non-null, it must be a pointer to memory associated with the given device, and it must not be managed by another `device_allocator`. The memory referenced by `device_memory` must be at least `sz_in_bytes` bytes in size and `sz_in_bytes` must be non-zero. Constructs an active `device_allocator<Device>` to manage the given `device_memory` as a GPU device segment, as if by calling:

device_allocator<Device>(device_object,sz_in_bytes,device_memory) and returns the result. The contents of device_memory remain unchanged.

16      *Exceptions:* May throw upcxx::bad_segment_alloc.

17      *UPC++ progress level:* user

## 16.2.6   Data Movement

```
1  template<typename T, memory_kind Kind1, memory_kind Kind2,
            typename Cx=/*...*/>
   RType copy(
       global_ptr<const T, Kind1> src, global_ptr<T, Kind2> dest,
       size_t count, Cx &&completions=operation_cx::as_future());
   template<typename T, memory_kind Kind, typename Cx=/*...*/>
   RType copy(
       T const *src, global_ptr<T, Kind> dest, size_t count,
       Cx &&completions=operation_cx::as_future());
   template<typename T, memory_kind Kind, typename Cx=/*...*/>
   RType copy(
       global_ptr<const T, Kind> src, T *dest, size_t count,
       Cx &&completions=operation_cx::as_future());
```

2   *Precondition:* T must be TriviallySerializable. The source and destination memory regions must not overlap. src and dest must not be null pointers, even if count is zero. src in the second variant and dest in the third variant must reference host memory.

3   Initiates an operation to transfer and store the count items of type T beginning at src to the memory beginning at dest. The values referenced in the [src,src+count) interval must not be modified until either source or operation completion is indicated.

4   Source- and operation-completion operations execute on the current (initiating) persona of the calling process. In the first and second variant, remote-completion operations execute on the master persona of the host process associated with the destination (i.e. dest.where()). In the third variant, remote-completion operations execute on the master persona of the calling process.

Completions:

5
- *Source*: Indicates completion of injection or internal buffering of the source values, signifying that the `src` buffer may be modified.

6
- *Remote*: Indicates completion of the transfer of the values, implying readiness of the target buffer `[dest,dest+count)`.

7
- *Operation*: Indicates completion of all aspects of the operation: the transfer and stores are complete.

8   *C++ memory ordering:* For LPC and RPC completions, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of the completion function.

9   *UPC++ progress level:* `internal`

16.3   ze_device oneAPI Interoperability

1   The `ze_device` memory kind class is used to manage access to a GPU device supporting the oneAPI Level-Zero device API. Previous sections described members of `ze_device` that are common to memory kind classes for all GPU devices. This section focuses on members specific to `ze_device` that enable interoperability with application code using the Level-Zero API and other portions of oneAPI. For details on the oneAPI specification, consult https://spec.oneapi.io/.

2   The oneAPI library organizes physical *Devices* (e.g. GPUs) into groups based on *Drivers*. The Level-Zero API names Devices and Drivers using opaque handles (`ze_device_handle_t` and `ze_driver_handle_t`, respectively). For convenience and uniformity with other Memory Kinds, each UPC++ process assigns a zero-based integral device index to each physical GPU accessible via Level Zero. `ze_device` provides static member functions for converting back-and-forth between this device ID and Level-Zero Device/Driver handles.

3   All oneAPI Device access is mediated through a *Context* associated with the corresponding Driver, which is represented using a `ze_context_handle_t`. For all Devices controlled by any given Level-Zero Driver, UPC++ requires that all `ze_device` segments encompass memory created using the same Level-Zero Driver Context. This same Level-Zero Context should also be shared by compute kernels accessing memory in the device segment.

4   Applications who create their own Level-Zero Context will need to supply that Context to UPC++ before opening the first `ze_device`. This can be done using the static member function `set_driver_context()`. For example:

```
1  #include <level_zero/ze_api.h>
2
3  device_allocator<ze_device>
4  my_make_allocator(size_t seg_size,
5                    ze_device_handle_t zeDevice,
6                    ze_context_handle_t zeContext) {
7    // share the oneAPI Driver Context with UPC++
8    ze_device::set_driver_context(zeContext, zeDevice);
9
10   // open the GPU and create device segment:
11   return make_gpu_allocator<ze_device>(seg_size,
12       ze_device::device_handle_to_device_id(zeDevice));
13 }
```

5 In the absence of calls to `ze_device::set_driver_context`, UPC++ defaults to creating a Level-Zero Context while activating the first `ze_device` on a given Driver. `ze_device` provides static member functions to retrieve the Level-Zero Context handle (as well as the Driver and Device handles). For example:

```
1  size_t seg_size = 2<<20;
2  // Create a GPU device segment, using automatic device selection:
3  auto gpu_alloc = make_gpu_allocator<ze_device>(seg_size);
4  // Query the ID of the opened device:
5  int id = gpu_alloc.device_id();
6  // Retrieve oneAPI handles corresponding to this device:
7  ze_device_handle_t zeDevice =
8      ze_device::device_id_to_device_handle(id);
9  ze_driver_handle_t zeDriver =
10     ze_device::device_id_to_driver_handle(id);
11 ze_context_handle_t zeContext =
12     ze_device::get_driver_context(zeDevice);
```

16.3.1   API Reference

```
1  class ze_device {
     using context_handle_t = /* see below */;
     using driver_handle_t = /* see below */;
     using device_handle_t = /* see below */;
     // ...
   };
```

2   Opaque types which are type-compatible with the Level-Zero handle types of similar name provided by oneAPI (i.e., `ze_context_handle_t`, `ze_driver_handle_t` and `ze_device_handle_t`, respectively).

3   *The remainder of this section uses* `ze_device`*-scoped type names without qualification, for clarity of presentation.*

4
```
static id_type ze_device :: device_handle_to_device_id (
        device_handle_t device_handle );
```

5   If `device_handle` corresponds to a Level-Zero Device known to the UPC++ library, returns the corresponding UPC++-defined integral device ID in the range `[0,ze_device::device_n())`.

6   Otherwise, returns `ze_device::invalid_device_id`.

7
```
static device_handle_t
    ze_device :: device_id_to_device_handle ( id_type device_id );
static driver_handle_t
    ze_device :: device_id_to_driver_handle ( id_type device_id );
```

8   *Precondition:* `device_id` must be in the range `[0,ze_device::device_n())`.

9   `device_id_to_device_handle` returns the Level-Zero Device handle corresponding to the device identified by `device_id`.

10   `device_id_to_driver_handle` returns the Level-Zero Driver handle corresponding to the device identified by `device_id`.

11
```
static context_handle_t
    ze_device :: get_driver_context ( driver_handle_t driver_handle );
static context_handle_t
    ze_device :: get_driver_context ( device_handle_t device_handle =
                                        device_handle_t ());
```

12   Returns a handle to the Level-Zero Context associated with the UPC++ library's access to all Level-Zero Devices encompassed by the Driver identified by the argument. The `driver_handle` overload identifies the Driver in question directly. The `device_handle` overload indicates the Driver encompassing the referenced Device (or encompassing the Device with ID 0 if the argument is defaulted).

13   If no Context has yet been associated with the Driver in question, then a new Context is created and associated as a side-effect of this call.

14   Callers shall not invoke `zeContextDestroy()` on the returned Context handle.

```
15  static void
    ze_device::set_driver_context(context_handle_t context_handle,
                                  driver_handle_t driver_handle);
    static void
    ze_device::set_driver_context(context_handle_t context_handle,
                                  device_handle_t device_handle =
                                                  device_handle_t());
```

16   *Precondition:* This process has neither called `get_driver_context()` for a Driver identified by the arguments, nor opened any `ze_device` encompassed by that Driver. The Context named by `context_handle` must have been created with visibility to all devices and any sub-devices supported by the Driver instance.

17   Establishes `context_handle` as the Level-Zero Driver Context to be used for all `ze_device` segments on Devices associated with the Driver identified by the second argument. The `driver_handle` overload identifies the Driver in question directly. The `device_handle` overload indicates the Driver encompassing the referenced Device (or encompassing the Device with ID 0 if the argument is defaulted).

18   Callers shall never invoke `zeContextDestroy()` on the provided Context.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi:10.1109/IEEESTD.2008.4610935.

[2] *ISO/IEC 14882:2011 Information technology - Programming Languages - C++*. 2011. URL: https://www.iso.org/standard/50372.html.

[3] *ISO/IEC 14882:2014 Information technology - Programming Languages - C++*. 2014. URL: https://www.iso.org/standard/64029.html.

[4] *ISO/IEC 14882:2017 Information technology - Programming Languages - C++*. 2017. URL: https://www.iso.org/standard/68564.html.

[5] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. UPC++: A High-Performance Communication Framework for Asynchronous Computation. In *Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium*, IPDPS. IEEE, 2019. doi:10.25344/S4V88H.

[6] Dan Bonachea and Paul H. Hargrove. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18)*, volume 11882 of *LNCS*. Springer, October 2018. doi:10.25344/S4QP4W.

[7] Amir Kamil and Dan Bonachea. Optimization of asynchronous communication operations through eager notifications. In *2021 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI+X (PAW-ATM)*, November 2021. doi:10.25344/S42C71.

[8] Daniel Waters, Colin A. MacLean, Dan Bonachea, and Paul H. Hargrove. Demonstrating UPC++/Kokkos interoperability in a heat conduction simulation. In *2021 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI+X (PAW-ATM)*, November 2021. doi:10.25344/S4630V.

[9] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS extension for C++. In *IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1105–1114, May 2014. doi:10.1109/IPDPS.2014.115.

Index