

UC San Diego

UC San Diego Previously Published Works

Title

UPC++ Specification v1.0, Draft 6

Permalink

<https://escholarship.org/uc/item/82094433>

Authors

Bachan, J
Baden, S
Bonachea, D
[et al.](#)

Publication Date

2018-03-26

DOI

10.2172/1430689

Peer reviewed

UPC++ Specification
v1.0 Draft 6

UPC++ Specification Working Group
upcxx-spec@googlegroups.com

March 26, 2018

Abstract

UPC++ is a C++11 library providing classes and functions that support Partitioned Global Address Space (PGAS) programming. We are revising the library under the auspices of the DOE's Exascale Computing Project, to meet the needs of applications requiring PGAS support. UPC++ is intended for implementing elaborate distributed data structures where communication is irregular or fine-grained. The UPC++ interfaces for moving non-contiguous data and handling memories with different optimal access methods are composable and similar to those used in conventional C++. The UPC++ programmer can expect communication to run at close to hardware speeds.

The key facilities in UPC++ are global pointers, that enable the programmer to express ownership information for improving locality, one-sided communication, both put/get and RPC, futures and continuations. Futures capture data readiness state, which is useful in making scheduling decisions, and continuations provide for completion handling via callbacks. Together, these enable the programmer to chain together a DAG of operations to execute asynchronously as high-latency dependencies become satisfied.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration

Early development of UPC++ was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Copyright

This manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

Legal Disclaimer

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

Contents

Contents	iii
1 Overview and Scope	1
1.1 Preliminaries	1
1.2 Execution Model	3
1.3 Memory Model	4
1.4 Organization of this Document	4
1.5 Conventions	4
1.6 Glossary	5
2 Init and Finalize	8
2.1 Overview	8
2.2 Hello World	8
2.3 API Reference	9
3 Global Pointers	11
3.1 Overview	11
3.2 API Reference	12
4 Storage Management	19
4.1 Overview	19
4.2 API Reference	19
5 Futures and Promises	23
5.1 Overview	23
5.2 The Basics of Asynchronous Communication	23
5.3 Working with Promises	25
5.4 Advanced Callbacks	26
5.5 Execution Model	28
5.6 Anonymous Dependencies	30

5.7	Lifetime and Thread Safety	31
5.8	API Reference	32
5.8.1	future	32
5.8.2	promise	35
6	Serialization	38
6.1	Class Serialization Interface	38
6.2	Serialization Concepts	39
6.3	Functions	41
6.4	Special Handling in Remote Procedure Calls	41
6.5	View-Based Serialization	42
6.6	API Reference	43
7	Completion	48
7.1	Overview	48
7.2	Completion Objects	50
7.2.1	Restrictions	52
7.2.2	Completion and Return Types	52
7.2.3	Default Completions	53
7.3	API Reference	53
8	One-Sided Communication	56
8.1	Overview	56
8.2	API Reference	56
8.2.1	Remote Puts	56
8.2.2	Remote Gets	58
9	Remote Procedure Call	60
9.1	Overview	60
9.2	Remote Hello World Example	61
9.3	API Reference	61
10	Progress	65
10.1	Overview	65
10.2	Restricted Context	66
10.3	Attentiveness	67
10.4	Thread Personas/Notification Affinity	67
10.5	API Reference	70
10.5.1	persona	71
10.5.2	persona_scope	74

10.5.3	Outgoing Progress	75
11	Atomics	76
11.1	Overview	76
11.2	API Reference	77
12	Teams	88
12.1	Overview	88
12.2	Local Teams	88
12.3	API Reference	89
12.3.1	team	89
12.3.2	team_id	91
12.3.3	Fundamental Teams	92
12.3.4	Collectives	93
13	Distributed Objects	96
13.1	Overview	96
13.2	Building Distributed Objects	97
13.3	Ensuring Distributed Existence	97
13.4	API Reference	98
14	Non-Contiguous One-Sided Communication	102
14.1	Overview	102
14.2	API Reference	103
14.2.1	Requirements on Iterators	103
14.2.2	Irregular Put	103
14.2.3	Irregular Get	105
14.2.4	Regular Put	106
14.2.5	Regular Get	108
14.2.6	Strided Put	109
14.2.7	Strided Get	111
15	Memory Kinds	113
A	Notes for Implementers	114
A.1	future_element_t and future_element_moved_t	114
A.2	future<T...>::when_all	115
A.3	to_future	116
A.4	future_invoke_result_t	116
	Bibliography	117

1 Chapter 1

2 Overview and Scope

3 1.1 Preliminaries

4 UPC++ is a C++11 library providing classes and functions that support Partitioned Global
5 Address Space (PGAS) programming. The project began in 2012 with a prototype AKA
6 V0.1, described in the IPDPS14 paper by *Zheng et al.* [3]. This document describes a
7 production version, V1.0, with the addition of several features and a new asynchronous
8 API.

9 Under the PGAS model, a distributed memory parallel computer is viewed abstractly
10 as a collection of *processing elements*, an individual computing resource, each with *local*
11 *memory* (see Fig. 1.1). A processing element is called a *rank* in UPC++. The execution
12 model of UPC++ is SPMD and the number of UPC++ ranks is fixed during program execution.

13 As with conventional C++ threads programming, ranks can access their respective
14 local memory via a pointer. However, the PGAS abstraction supports a global address
15 space, which is allocated in *shared segments* distributed over the ranks. A *global pointer*
16 enables the programmer to move data in the shared segments between ranks as shown in
17 Fig. 1.1. As with threads programming, references made via global pointers are subject to
18 race conditions, and appropriate synchronization must be employed.

19 UPC++ global pointers are fundamentally different from conventional C-style pointers. A
20 global pointer refers to a location in a shared segment. It cannot be dereferenced using the
21 \star operator, and it does not support conversions between pointers to base and derived types.
22 It also cannot be constructed by the address-of operator. On the other hand, UPC++ global
23 pointers *do* support some properties of a regular C pointer, such as pointer arithmetic and
24 passing a pointer by value.

25 Notably, global pointers are used in *one-sided* communication: bulk copying operations
26 (RMA) similar to *memcpy* but across ranks (Ch. 8), and in Remote Procedure Calls

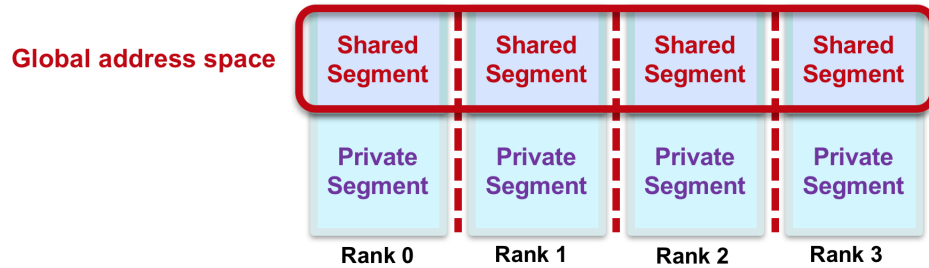


Figure 1.1: Abstract Machine Model of a PGAS program memory

1 (RPC, Ch. 9). RPC enables the programmer to ship functions to other ranks, which is
 2 useful in managing irregular distributed data structures. These ranks can push or pull
 3 data via global pointers. *Futures* and *Promises* (Ch. 5) are used to determine completion
 4 of communication or to provide handlers that respond to completion. Wherever possible,
 5 UPC++ will engage low-level hardware support for communication and this capability is
 6 crucial to UPC++’s support of *lightweight communication*.

7 UPC++’s design philosophy is to provide “close to the metal performance.” To meet this
 8 requirement, UPC++ imposes certain restrictions. In particular, non-blocking communica-
 9 tion is the default for nearly all operations defined in the API, and all communication is
 10 explicit. These two restrictions encourage the programmer to write code that is perfor-
 11 mant and make it more difficult to write code that is not. Conversely, UPC++ relaxes some
 12 restrictions found in models such as MPI; in particular, it does not impose an in-order
 13 delivery requirement between separate communication operations. The added flexibility
 14 increases the possibility of overlapping communication and scheduling it appropriately.

15 UPC++ also avoids non-scalable constructs found in models such as UPC. For example,
 16 it does not support shared distributed arrays or shared scalars. Instead, it provides dis-
 17 tributed objects, which can be used to similar ends (Ch. 13). Distributed objects are useful
 18 in solving the *bootstrapping problem*, whereby ranks need to distribute their local copies
 19 of global pointers to other ranks. Though UPC++ does not directly provide multidimen-
 20 sional arrays, applications that use UPC++ may define them. To this end, UPC++ supports
 21 non-contiguous data transfers: vector, indexed, and strided data (Ch. 14).

22 Because UPC++ does not provide separate concurrent threads to manage progress, UPC++
 23 must manage all progress inside active calls to the library. UPC++ has been designed with a
 24 policy against the use of internal operating system threads. The strengths of this approach
 25 are improved user-visibility into the resource requirements of UPC++ and better interoper-
 26 ability with software packages and their possibly restrictive threading requirements. The
 27 consequence, however, is that the user must be conscientious to balance the need for mak-
 28 ing progress against the application’s need for CPU cycles. Chapter 10 discusses subtleties

1 of managing progress and how an application can arrange for UPC++ to advance the state
2 of asynchronous communication.

3 Ranks may be grouped into teams (Ch. 12). A team can participate in collective
4 operations. Teams are also the interface that UPC++ uses to propagate the shared memory
5 capabilities of the underlying hardware and operating system and can let a programmer
6 reason about hierarchical processor-memory organization, allowing an application to reduce
7 its memory footprint. UPC++ supports atomic operations, currently on remote 32-bit and
8 64-bit integers. Atomics are useful in managing distributed queues, hash tables, and so
9 on. However, as explained in the discussion below on UPC++'s memory model, atomics are
10 split phased and not handled the same way as they are in C++11 and other libraries.

11 UPC++ will support memory kinds (Ch. 15), whereby the programmer can identify re-
12 gions of memory requiring different access methods or having different performance prop-
13 erties, such as device memory. Since memory kinds will be implemented in Year 2, we will
14 defer their detailed discussion until next year.

15 1.2 Execution Model

16 The UPC++ internal state contains, for each rank, internal unordered queues that are man-
17 aged for the user. The UPC++ progress engine scans these queues for operations initiated by
18 this rank, as well as externally generated operations that target this rank. The progress en-
19 gine is active inside UPC++ calls only and is quiescent at other times, as there are no threads
20 or background processes executing inside UPC++. This passive stance permits UPC++ to be
21 driven by any other execution model a user might choose. This universality does place a
22 small burden on the user: calling into the `progress` function. UPC++ relies on the user to
23 make periodic calls into the `progress` function to ensure that UPC++ operations are com-
24 pleted. `progress` is the mechanism by which the user loans UPC++ a thread of execution
25 to perform operations that target the given rank. The user can determine that a specific
26 operation completes by checking the status of its associated `future`, or by attaching a
27 completion handler to the operation.

28 UPC++ presents a *thread-aware* programming model. It assumes that only one thread
29 of execution is interacting with any UPC++ object. The abstraction for thread-awareness
30 in UPC++ is the *persona*. A `future` produced by a thread of execution is associated with
31 its persona, and transferring the `future` to another thread must be accompanied by trans-
32 ferring the underlying persona. Each rank has a *master persona*, initially attached to the
33 thread that calls `init`. Some UPC++ operations, such as `barrier`, require a thread to have
34 exclusive access to the master persona to call them. Thus, the programmer is responsible
35 for ensuring synchronized access to both personas and memory, and that access to shared
36 data does not interfere with the internal operation of UPC++.

1.3 Memory Model

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, atomic 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.

UPC++ differs from MPI in that it doesn't guarantee in-order delivery. For example, if we overlap two successive RPC operations involving the same source and destination rank, we cannot say which one completes first.

1.4 Organization of this Document

This specification is intended to be a normative reference - a Programmer's Manual is forthcoming. For the purposes of understanding the key ideas in UPC++, we recommend that the novice reader skip Chapter 10 (Progress) and the advanced topics related to futures, personas, and continuation-based communication.

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 storage allocation. Since UPC++ supports asynchronous communication only, UPC++ provides futures and promises (Ch. 5) to manage control flow and completion. Chapters 8 and 9 describe the two forms of asynchronous one-sided communication, `rput/rget` and RPC, respectively. Chapter 10 discusses progress. Chapter 11 discusses atomic operations. Chapter 12 discusses teams, which are a means of organizing UPC++ ranks. Chapter 13 discusses distributed objects. Chapter 14 discusses non-contiguous data transfers. Chapter 15 discusses memory kinds.

1.5 Conventions

1. C++ language keywords are in the color `mocha`.
2. UPC++ terms are set in the color `bright blue` except when they appear in a synopsis framebox.
3. All functions are declared `noexcept` unless specifically called out.
4. All entities are in the `upcxx` namespace unless otherwise qualified.

1.6 Glossary

Affinity. A binding of each location in a shared segment to a particular rank (generally the rank which allocated that shared object). Every byte of shared memory has affinity to exactly one rank (at least logically).

C++ Concepts. E.g. `TriviallyCopyable`. This document references C++ Concepts as defined in the C++14 standard [2] when specifying the semantics of types. However, compliant implementations are still possible within a compiler adhering to the earlier C++11 standard [1].

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

A collective operation need not provide any actual synchronization between ranks, 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 ranks within collective operations, but programs must not rely upon the presence or absence of such unspecified synchronization for correctness.

DefinitelySerializable. (6) A C++ type that is either `DefinitelyTriviallySerializable`, or for which there is a user-supplied implementation of the visitor function `serialize`.

DefinitelyTriviallySerializable. (6) A C++ type that is either `TriviallyCopyable` and has no user-supplied implementation of the visitor function, or for which the `is_definitely_trivially_serializable` trait is specialized to have a member value that is true.

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 ranks. 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,

1 and it is through the promise that the results of the operation are supplied and
2 its dependencies fulfilled.

3 **Global pointer.** (3) The primary way to address memory in a shared memory seg-
4 ment of a UPC++ program. Global pointers can themselves be stored in shared
5 memory or otherwise passed between ranks and retain their semantic meaning
6 to any rank.

7 **Local.** (12.2) Refers to an object or reference with affinity to a rank in the local
8 team.

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

13 **Persona.** (10.4) The abstraction for thread-awareness in UPC++. A UPC++ persona
14 object represents a collection of UPC++-internal state usually attributed to a sin-
15 gle thread. By making it a proper construct, UPC++ allows a single OS thread
16 to switch between multiple application-defined roles for processing notifications.
17 Personas act as the receivers for notifications generated by the UPC++ runtime.

18 **Private object.** An object outside the shared space that can be accessed only by
19 the rank that owns it (e.g. an object on the program stack).

20 **Progress.** (10) The means by which the application allows the UPC++ runtime to
21 advance the state of outstanding operations initiated by this or other ranks, to
22 ensure they eventually complete.

23 **Rank.** An OS process that is a member of a UPC++ parallel job execution. UPC++
24 uses a SPMD execution model, and the number of ranks is fixed during a given
25 program execution. The placement of ranks across physical processors or NUMA
26 domains is implementation-dependent.

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

30 **Remote.** Refers to an object or reference whose affinity is not local to the current
31 rank.

32 **Remote Procedure Call.** A communication operation that injects a function call
33 invocation into the execution stream of another rank. These injections are
34 one-sided, meaning the target rank need not explicitly expect the incoming op-
35 eration or perform any specific action to receive it, aside from invoking UPC++
36 progress.

1 **Source completion.** The condition where a communication operation initiated by
2 the current rank has advanced to a point where serialization of the local source
3 memory regions for the operation has occurred, and the contents of those re-
4 gions can be safely overwritten or reclaimed without affecting the behavior of the
5 ongoing operation. Source completion does not generally imply operation com-
6 pletion, and other effects of the operation (e.g., updating destination memory
7 regions, or delivery to a remote rank) may still be in-progress.

8 **Shared segment.** A region of storage associated with a particular rank that is used
9 to allocate shared objects that are accessible by any rank.

10 **Team.** A UPC++ object representing an ordered set of ranks.

11 **Thread (or OS thread).** An independent stream of executing instructions with
12 private state. A rank process may contain many threads (created by the appli-
13 cation), and each is associated with at least one persona.

14 **Serializable.** (6) A C++ type that is either DefinitelySerializable or TriviallySeri-
15 alizable.

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

1 Chapter 2

2 Init and Finalize

3 2.1 Overview

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

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

21 2.2 Hello World

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

```

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 UPC++ ranks?
9         << " my rank: " << upcxx::rank_me() // which rank am I?
10        << std::endl;
11
12    upcxx::finalize(); // finalize UPC++
13    return 0;
14 }

```

Figure 2.1: *HelloWorld.cpp* program

1 different processes may even be interleaved.

2 2.3 API Reference

```
3 void init();
```

4 *Preconditions:* Called collectively by all processes in the parallel job. Call-
5 ing thread must have the master persona (§10.5.1) if UPC++ is in an already-
6 initialized state.

7 If there have been no previous calls to `init`, or if all previous calls to `init` have
8 had matching calls to `finalize`, then this routine initializes the UPC++ library.
9 Otherwise, leaves the library's state as is. Upon return, the calling thread will
10 be attached to the master persona (§10.5.1).

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

```
12 void finalize();
```

13 *Preconditions:* Called collectively by all processes in the parallel job. Call-
14 ing thread must have the master persona (§10.5.1), and UPC++ must be in an
15 already-initialized state.

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

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

13 *UPC++ progress level: user*

1 Chapter 3

2 Global Pointers

3 3.1 Overview

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

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

17 Global pointers have the following guarantees:

- 18 1. A `global_ptr<T>` is only valid if it is the null global pointer, it references a valid
19 object, or it represents one element past the end of a valid array or non-array object.
- 20 2. Two global pointers compare equal if and only if they reference the same object, one
21 past the end of the same array or non-array object, or are both null.
- 22 3. Equality of global pointers corresponds to observational equality, meaning that two
23 global pointers which compare equal will produce equivalent behavior when inter-
24 changed.

25 These facts become important given that UPC++ allows two ranks which are local to
26 each other to map the same memory into their own virtual address spaces but possibly

1 with different virtual addresses. They also ensure that a global pointer can be viewed from
2 any rank to mean the same thing without need for translation.

3 **3.2 API Reference**

```
4 using intrank_t = /* implementation-defined */;
```

5 An implementation-defined signed integer type that represents a UPC++ rank
6 ID.

```
7 template<typename T>  
8 struct global_ptr;
```

9 C++ Concepts: DefaultConstructible, TriviallyCopyable, TriviallyDestructible,
10 EqualityComparable, LessThanComparable, hashable

11 UPC++ Concepts: DefinitelyTriviallySerializable

12 T must not have any cv qualifiers: `std::is_const<T>::value` and
13 `std::is_volatile<T>::value` must both be false.

```
14 template<typename T>  
15 struct global_ptr {  
16     using element_type = T;  
17     // ...  
18 };
```

19 Member type that is an alias for the template parameter T.

```
20 template<typename T>  
21 global_ptr<T>::global_ptr(T* ptr);
```

22 *Precondition:* `ptr` must be either null or an address in the shared segment (Ch.
23 4) of a rank in the local team (§12.2)

24 Constructs a global pointer corresponding to the given raw pointer. This con-
25 structor must be called explicitly.

26 *UPC++ progress level:* none

```
27 template<typename T>  
28 global_ptr<T>::global_ptr(std::nullptr_t = nullptr);
```

1 Constructs a global pointer corresponding to a null pointer.

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

3 *UPC++ progress level: none*

```
4 template<typename T>  
5 global_ptr<T>::~~global_ptr();
```

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

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

9 *UPC++ progress level: none*

```
10 template<typename T>  
11 bool global_ptr<T>::is_local() const;
```

12 Returns whether or not the calling rank has load/store access to the memory
13 referenced by this pointer. Returns true if this is a null pointer, regardless of
14 the context in which this query is called.

15 *UPC++ progress level: none*

```
16 template<typename T>  
17 bool global_ptr<T>::is_null() const;
```

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

21 *UPC++ progress level: none*

```
22 template<typename T>  
23 T* global_ptr<T>::local() const;
```

24 *Precondition: this->is_local()*

25 Converts this global pointer into a raw pointer.

26 *UPC++ progress level: none*

```
1  template<typename T>
2  intrank_t global_ptr<T>::where() const;
```

3 Returns the rank in team `world()` with affinity to the T object pointed-to by
4 this global pointer. The return value for `where()` on a null global pointer is
5 an implementation-defined value. This query is purely a function of the global
6 pointer instance, it is not affected by the context in which it is called.

7 *UPC++ progress level: none*

```
8  template<typename T>
9  global_ptr<T> global_ptr<T>::operator+(std::ptrdiff_t diff) const;
10 template<typename T>
11 global_ptr<T> operator+(std::ptrdiff_t diff, global_ptr<T> ptr);
```

```
12 template<typename T>
13 global_ptr<T>& global_ptr<T>::operator+=(std::ptrdiff_t diff);
```

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

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

23 `operator+=` modifies the `global_ptr` in-place and return a reference to itself
24 after the operation.

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

27 *UPC++ progress level: none*

```
28 template<typename T>
29 global_ptr<T> global_ptr<T>::operator-(std::ptrdiff_t diff) const;
```

```
30 template<typename T>
31 global_ptr<T>& global_ptr<T>::operator-=(std::ptrdiff_t diff);
```

Precondition: 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.

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.

`operator--` modifies the `global_ptr` in-place and return a reference to itself after the operation.

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

UPC++ progress level: none

```
15 template<typename T>
16 std::ptrdiff_t global_ptr<T>::operator-(global_ptr<T> rhs) const;
```

Precondition: 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.

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

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

UPC++ progress level: none

```
26 template<typename T>
27 global_ptr<T>& global_ptr<T>::operator++();
```

Precondition: the global pointer must be pointing to an element of an array or to a non-array object

Modifies this pointer to have the value `*this + 1` and returns a reference to this pointer.

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

UPC++ progress level: none

```
1  template<typename T>
2  global_ptr<T> global_ptr<T>::operator++(int);
```

3 *Precondition:* the global pointer must be pointing to an element of an array or
4 to a non-array object

5 Modifies this pointer to have the value `*this + 1` and returns a copy of the
6 original pointer.

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

9 *UPC++ progress level: none*

```
10 template<typename T>
11 global_ptr<T>& global_ptr<T>::operator--();
```

12 *Precondition:* the global pointer must either be pointing to the `i`th element of
13 an array, where `i >= 1`, or one element past the end of an array or a non-array
14 object

15 Modifies this pointer to have the value `*this - 1` and returns a reference to
16 this pointer.

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

19 *UPC++ progress level: none*

```
20 template<typename T>
21 global_ptr<T> global_ptr<T>::operator--(int);
```

22 *Precondition:* the global pointer must either be pointing to the `i`th element of
23 an array, where `i >= 1`, or one element past the end of an array or a non-array
24 object

25 Modifies this pointer to have the value `*this - 1` and returns a copy of the
26 original pointer.

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

29 *UPC++ progress level: none*

```

1  template<typename T>
2  bool global_ptr<T>::operator==(global_ptr<T> rhs) const;
3  template<typename T>
4  bool global_ptr<T>::operator!=(global_ptr<T> rhs) const;
5  template<typename T>
6  bool global_ptr<T>::operator<(global_ptr<T> rhs) const;
7  template<typename T>
8  bool global_ptr<T>::operator<=(global_ptr<T> rhs) const;
9  template<typename T>
10 bool global_ptr<T>::operator>(global_ptr<T> rhs) const;
11 template<typename T>
12 bool global_ptr<T>::operator>=(global_ptr<T> rhs) const;

```

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

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

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

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

29 All other comparisons result in an unspecified value.

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

32 *UPC++ progress level: none*

```

33 namespace std {
34     template<typename T>
35     struct less<global_ptr<T>>;
36     template<typename T>

```



```
1  struct less_equal<global_ptr<T>>;
2  template<typename T>
3  struct greater<global_ptr<T>>;
4  template<typename T>
5  struct greater_equal<global_ptr<T>>;
6  template<typename T>
7  struct hash<global_ptr<T>>;
8  }
```

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

15 *UPC++ progress level: none*

```
16 template<typename T>
17 std::ostream& operator<<(std::ostream &os, global_ptr<T> ptr);
```

18 Inserts an implementation-defined character representation of `ptr` into the out-
19 put stream `os`. This function can be called on any valid global pointer, and the
20 textual representation of two objects of type `global_ptr<T>` is identical if and
21 only if the two global pointers compare equal.

22 *UPC++ progress level: none*

```
23 template<typename T, typename U>
24 global_ptr<T> reinterpret_pointer_cast(global_ptr<U> ptr);
```

25 *Precondition:* the expression `reinterpret_cast<T*>((U*)nullptr)` must be
26 well formed

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

30 If `rp` is the raw pointer of `ptr`, then the raw pointer of the result is constructed
31 by `reinterpret_cast<T*>(rp)`.

32 *UPC++ progress level: none*

1 Chapter 4

2 Storage Management

3 4.1 Overview

4 UPC++ provides two flavors of storage allocation involving the shared segment. The pair
5 of functions `new_` and `delete_` will call the class constructors and destructors, respectively,
6 as well as allocate and deallocate memory from the shared segment. The pair `allocate`
7 and `deallocate` allocate and deallocate dynamic memory from the shared segment, but
8 do not call C++ constructors or destructors. A user may call these functions directly, or
9 use placement new, or other memory management practices.

10 4.2 API Reference

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

13 *Precondition:* `T(args...)` must be a valid call to a constructor for `T`.

14 Allocates space for an object of type `T` from the shared segment of the current
15 rank. If the allocation succeeds, returns a pointer to the start of the allocated
16 memory, and the object is initialized by invoking the constructor `T(args...)`.
17 If the allocation fails, throws `std::bad_alloc`.

18 *Exceptions:* May throw `std::bad_alloc` or any exception thrown by the call
19 `T(args...)`.

20 *UPC++ progress level:* none

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

1 *Precondition:* `T(args...)` must be a valid call to a constructor for `T`.

2 Allocates space for an object of type `T` from the shared segment of the current
3 rank. If the allocation succeeds, returns a pointer to the start of the allocated
4 memory, and the object is initialized by invoking the constructor `T(args...)`.
5 If the allocation fails, returns a null pointer.

6 *Exceptions:* May throw any exception thrown by the call `T(args...)`.

7 UPC++ progress level: **none**

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

10 *Precondition:* `T` must be `DefaultConstructible`.

11 Allocates space for an array of `n` objects of type `T` from the shared segment of
12 the current rank. If the allocation succeeds, returns a pointer to the start of
13 the allocated memory, and the objects are initialized by invoking their default
14 constructors. If the allocation fails, throws `std::bad_alloc`.

15 *Exceptions:* May throw `std::bad_alloc` or any exception thrown by the call
16 `T()`. If an exception is thrown by the constructor for `T`, then previously initial-
17 ized elements are destroyed in reverse order of construction.

18 UPC++ progress level: **none**

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

21 *Precondition:* `T` must be `DefaultConstructible`.

22 Allocates space for an array of `n` objects of type `T` from the shared segment of
23 the current rank. If the allocation succeeds, returns a pointer to the start of
24 the allocated memory, and the objects are initialized by invoking their default
25 constructors. If the allocation fails, returns a null pointer.

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

29 UPC++ progress level: **none**

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

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

4 If `g` is not a null pointer, invokes the destructor on the given object and deal-
 5 locates the storage allocated to it. Does nothing if `g` is a null pointer.

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

7 *UPC++ progress level:* none

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

10 *Precondition:* `T` must be Destructible. `g` must be either a null pointer or a non-
 11 deallocated pointer that resulted from a call to `new_array<T>` on the current
 12 rank.

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

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

17 *UPC++ progress level:* none

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

20 *Precondition:* `alignment` is a valid alignment. `size` must be an integral mul-
 21 tiple of `alignment`.

22 Allocates `size` bytes of memory from the shared segment of the current rank,
 23 with alignment as specified by `alignment`. If the allocation succeeds, returns
 24 a pointer to the start of the allocated memory, and the allocated memory is
 25 uninitialized. If the allocation fails, returns a null pointer.

26 *UPC++ progress level:* none

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

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

2 Allocates enough space for `n` objects of type `T` from the shared segment of
3 the current rank, with the memory aligned as specified by `alignment`. If the
4 allocation succeeds, returns a pointer to the start of the allocated memory, and
5 the allocated memory is uninitialized. If the allocation fails, returns a null
6 pointer.

7 *UPC++ progress level:* `none`

8 `void deallocate(void* p);`

9 *Precondition:* `p` must be either a null pointer or a non-deallocated pointer that
10 resulted from a call to the first form of `allocate` on the current rank.

11 Deallocates the storage previously allocated by a call to `allocate`. Does noth-
12 ing if `p` is a null pointer.

13 *UPC++ progress level:* `none`

14 `template<typename T>`

15 `void deallocate(global_ptr<T> g);`

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

19 Deallocates the storage previously allocated by a call to `allocate`. Does noth-
20 ing if `g` is a null pointer. Does not invoke the destructor for `T`.

21 *UPC++ progress level:* `none`

1 Chapter 5

2 Futures and Promises

3 5.1 Overview

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

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

21 5.2 The Basics of Asynchronous Communication

22 A programmer can invoke a non-blocking operation to be serviced by another rank, such
23 as a one-sided get operation (Ch. 8) or a remote procedure call (Ch. 9). Such an operation

¹Another mechanism, persona-targeted continuations, is discussed in §10.4.

1 creates an implicit promise and returns an associated future object to the user. When the
2 operation completes, the future becomes ready, and it can be used to access the results.
3 The following demonstrates an example using a remote get (see Ch. 10 on how to make
4 progress with UPC++):

```
5 global_ptr<double> ptr = /* obtain some remote pointer */;  
6 future<double> fut = rget(ptr);           // initiate a remote get  
7 // ...call into upcxx::progress() elided...  
8 if (fut.ready()) {                       // check for readiness  
9     double value = fut.result();         // retrieve result  
10    std::cout << "got: " << value << '\n'; // use result  
11 }
```

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

```
16 global_ptr<double> ptr = /* obtain some remote pointer */;  
17 future<double> fut = rget(ptr);           // initiate a remote get  
18 fut.wait();                             // wait for completion  
19 double value = fut.result();             // retrieve result  
20 std::cout << "got: " << value << '\n'; // use result
```

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

```
25 global_ptr<double> ptr = /* obtain some remote pointer */;  
26 auto fut =  
27 rget(ptr).then( // initiate a remote get and register a callback  
28 // lambda callback function  
29 [](double value) {  
30     std::cout << "got: " << value << '\n'; // use result  
31 }  
32 );
```

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

36 A future can also represent the completion of a combination of several non-blocking
37 operations. Unlike the standard C++ future, `upcxx::future` is a variadic template, encapsulating
38 an arbitrary number of result values that can come from different operations. The
39 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);
```

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

7 5.3 Working with Promises

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

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

```
20 global_ptr<int> ptrs[10] = /* some remote pointers */;  
21 // create a promise with no results  
22 // the dependency count starts at one  
23 promise<> prom;  
24  
25 // do 10 puts, registering each of them on the promise  
26 for (int i = 0; i < 10; i++) {  
27     // rput implicitly registers itself on the given promise  
28     rput(i, ptrs[i], operation_cx::as_promise(prom));  
29 }  
30  
31 // fulfill initial anonymous dependency, since registration is done  
32 future<> fut = prom.finalize();  
33  
34 // wait for the rput operations to complete  
35 fut.wait();
```


1 5.4 Advanced Callbacks

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

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

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

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

```
18 future<int, double> source = /* obtain a future */;  
19 future<double> result = source.then(  
20     [](int x, double y) {  
21         return x + y;  
22     }  
23 );
```

24 In this example, `source` is a future representing an `int` and a `double` value. The
25 argument of the call to `then` must be a function object that can be called on these values.
26 Here, we use a lambda function that takes in an `int` and a `double`. The call to `then`
27 returns a future that represents the result of calling the argument of `then` on the values
28 contained in `source`. Since the lambda function above returns a `double`, the result of `then`
29 is a `future<double>` that will hold the `double`'s value when it is ready.

30 There is also another case, when the callback returns a future, rather than some non-
31 future type. In previous case, the result of `then()` is obtained by wrapping return type
32 inside a future. In this case, this step is not needed, as we are already returning a future.
33 Thus, the result of the call to `then` has the same type as the return type of the callback.
34 However, there is an important difference: the result is a future, which may or may not be
35 ready. In the first case, it is the returned non-future value that may or may or may not
36 be ready. This subtle difference, allows the UPC++ programmer to chain the results of one
37 asynchronous operation into the inputs of the next, to arbitrary degree of nesting.

```
38 future<int, double> source = /* obtain a future */;
```

```

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

```

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

```

11 global_ptr<int> remote_array = /* some remote array */;
12
13 // retrieve remote_array[0]
14 future<int> elt0 = rget(remote_array);
15
16 // retrieve remote_array[remote_array[0]]
17 future<int> elt_indirect = elt0.then(
18     [=](int index) {
19         return rget(remote_array + index);
20     }
21 );

```

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 post-transformation value. For example, we can future transform the value of `elt_indirect` above as follows:

```

26 future<int> elt_indirect_squared = elt_indirect.then(
27     [](int value) {
28         return value * value;
29     }
30 );

```

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:

```

36 future<int> value1 = /* ... */;
37 future<double> value2 = /* ... */;
38
39 future<int, double> combined = when_all(value1, value2);

```

```
1 future<double> result = combined.then(  
2     [](int x, double y) {  
3         return x + y;  
4     }  
5 );
```

6 In the more general case, we may need to combine heterogeneous mixtures of future and
7 non-future types. The `to_future` function template wraps a non-future value in a future
8 while leaving future values unchanged. Thus, we can use `when_all` along with `to_future`
9 to construct a single future that represents the combination of both future and non-future
10 values:

```
11 future<int> value1 = /* ... */;  
12 double value2 = /* ... */;  
13  
14 future<int, double> combined = when_all(to_future(value1),  
15                                         to_future(value2));  
16 future<double> result = combined.then(  
17     [](int x, double y) {  
18         return x + y;  
19     }  
20 );
```

21 The results of a future can be obtained, if it is ready, as a `std::tuple` using the
22 `result_tuple` member function of a future. Individual components can be retrieved by
23 value with the `result` member function template or by r-value reference with `result_moved`.
24 Unlike with `std::get`, it is not a compile-time error to use an invalid index with `result`
25 or `result_moved`; instead, the return type is `void` for an invalid index. This simplifies
26 writing generic functions on futures, such as the following C++14-compliant definition of
27 `wait`:

```
28 template<typename ...T>  
29 auto future<T...>::wait() {  
30     while (!ready()) {  
31         progress();  
32     }  
33     return result();  
34 }
```

35 5.5 Execution Model

36 Futures have the capability to express dataflow/task-based programming, and other soft-
37 ware frameworks provide thread-level parallelism by considering each callback to be a task

1 that can be run in an arbitrary worker thread. This is not the case in UPC++. In order
2 to maximize performance, our approach to futures is purposefully ambivalent to issues of
3 concurrency. A UPC++ implementation is allowed to take action as if the current thread is
4 the only one that needs to be accounted for. This gives rise to a natural execution policy:
5 callbacks registered against futures are always executed as soon as possible by the thread
6 that discovers them. There are exactly two scenarios in which this may happen:

- 7 1. When a promise is fulfilled.
- 8 2. A callback is registered onto a ready future using the `then` member function.

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

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

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

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

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

1 5.6 Anonymous Dependencies

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

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

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

14 The argument to `require_anonymous` must be strictly greater than the negation of
15 the promise's dependency count, so that a call to `require_anonymous` never causes the
16 dependency count to reach zero, putting the promise in the fulfilled state. In the example
17 above, the argument must be greater than -1, and the given argument of 10 is valid.

18 Anonymous dependencies can be fulfilled by calling the `fulfill_anonymous` member
19 function:

```
20 for (int i = 0; i < 5; i++) {  
21     pro.fulfill_anonymous(i);  
22 } // dependency count is now 1
```

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

```
25 pro.fulfill_result(3, 4.1); // dependency count is now 0  
26 assert(pro.get_future().ready());
```

27 Both empty and non-empty promises can be used to track anonymous dependencies. A
28 UPC++ operation that operates on a promise *always* increments its dependency count upon
29 invocation, as if by calling `require_anonymous(1)` on the promise. After the operation
30 completes², if the completion produces values of type `T...`, then the values are supplied
31 to the promise through a call to `fulfill_result`. Otherwise, the completion is signaled
32 by fulfilling an anonymous dependency through a call to `fulfill_anonymous(1)`.

33 The rationale for this behavior is to free the user from having to manually increment the
34 dependency count before calling an operation on a promise; instead, UPC++ will implicitly
35 perform this increment. This leads to the pattern, shown at the beginning of this chapter,

²The notification will occur during user-level progress of the persona that initiates the operation. See Ch. 10 for more details.

1 of registering operations on a promise and then finalizing the promise to take it out of
2 registration mode:

```

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

```

15 A user familiar with UPC++ V0.1 will observe that empty promises subsume the ca-
16 pabilities of events in UPC++ V0.1. In addition, they can take part in all the machinery
17 of promises, futures, and callbacks, providing a much richer set of capabilities than were
18 available in V0.1.

19 5.7 Lifetime and Thread Safety

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

27 Though UPC++ does not prescribe a specific management strategy, the semantics of
28 futures and promises are analogous to those of standard C++11 smart pointers. As with
29 `std::shared_ptr`, a future may be freely copied, and both the original and the copy
30 represent the same state and are associated with the same promise. Thus, if one copy
31 of a future becomes ready, then so will the other copies. On the other hand, a promise
32 can be mutated by the user through its member functions, so allowing a promise to be
33 copied would introduce the issue of aliasing. Instead, we adopt the same non-copyable, yet
34 movable, semantics for a promise as `std::unique_ptr`.

35 Given that UPC++ futures and promises are already thread-unaware to allow the ex-
36 ecution strategy to be straightforward and efficient, UPC++ also makes no thread safety
37 guarantees about internal state management. This enables creation of copies of a future

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

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

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

25 5.8 API Reference

26 *UPC++ progress level for all functions in this chapter (unless otherwise noted) is: none*

27 5.8.1 future

```
28 template<typename ...T>  
29 class future;
```

30 C++ Concepts: DefaultConstructible, CopyConstructible, CopyAssignable,
31 Destructible

32 The types in T... must not be void.

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

1 Constructs a future that will never become ready.

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

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

5 Destroys this future object.

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

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

9 Constructs a trivially ready future from the given values.

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

12 Returns true if the future's result values have been supplied to it.

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

15 *Precondition:* `this->ready()`

16 Retrieves the tuple of result values for this future.

```
17 template<typename ...T>  
18 template<int I=0>  
19 future_element_t<I, future<T...>>  
20 future<T...>::result() const;
```

21 *Precondition:* `this->ready()`

22 Retrieves the I^{th} component (defaults to first) from the future's results tuple.
23 The return type is `void` if I is an invalid index. Otherwise it is of type U , where
24 U is the I^{th} component of T .


```
1  template<typename ...T>
2  template<int I=0>
3  future_element_moved_t<I, future<T...>>
4  future<T...>::result_moved();
```

5 *Precondition:* `this->ready()`

6 Retrieves the I^{th} component (defaults to first) from the future's results tuple as
7 an r-value reference, as if by calling `std::move` on the component. The return
8 type is `void` if I is an invalid index. Otherwise it is of type `U&&`, where U is
9 the I^{th} component of T . *Caution: this operation permits mutation of the value,*
10 *via an rvalue reference which could be observed by further calls that return the*
11 *result(s) of a future.*

```
12  template<typename ...T>
13  template<typename Func>
14  future_invoke_result_t<Func, T...>
15  future<T...>::then(Func func);
```

16 *Preconditions:* The call `func()` must not throw an exception.

17 Returns a new future representing the return value of the given function object
18 `func` when invoked on the results of this future as its argument list. If `func`
19 returns a future, then the result of `then` will be a semantically equivalent future,
20 except that it will be in a non-ready state before `func` executes. If `func` does
21 not return a future, then the return value of `then` is a future that encapsulates
22 the result of `func`, and this future will also be in a non-ready state before `func`
23 executes. If the return type of `func` is `void`, then the return type of `then` is
24 `future<>`.

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

- 26 • Immediately before `then` returns if this future is in the ready state.
- 27 • During a promise fulfillment which would directly or indirectly make this
28 future transition to the ready state.

```
29  template<typename ...T>
30  future_element_t<0, future <T...>> future<T...>::wait();
```

31 Waits for the future by repeatedly attempting UPC++ user-level progress and
32 testing for readiness. See Ch. 10 for a discussion of progress. The return value
33 is the same as that produced by calling `result()` on the future.

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

```
1  template<typename ...Futures>
2  future<CTypes...> when_all(Futures ...fs);
```

3 Given a variadic list of futures as arguments, constructs a future representing
4 the readiness of all arguments. The results tuple of this future will be the
5 concatenated results tuples of the arguments. The type parameters of the
6 returned object (`CTypes...`) is the ordered concatenation of the type parameter
7 lists of the types in `Futures...`. If `Futures...` is empty, then the result is a
8 trivially ready `future<>`.

```
9  template<typename T>
10 future<CTypes...> to_future(T future_or_value);
```

11 Constructs a future that encapsulates the value represented by `future_or_`
12 `value`. If `T` is of type `future<U...>`, then `CTypes...` is the same as `U...`, and
13 the returned future is a copy of `future_or_value`. If `T` is not a future, then
14 `CTypes...` is `T`, and the function returns a ready future whose encapsulated
15 value is `future_or_value`.

16 5.8.2 promise

```
17 template<typename ...T>
18 class promise;
```

19 C++ Concepts: `DefaultConstructible`, `MoveConstructible`, `MoveAssignable`,
20 `Destructible`

21 The types in `T...` must not be `void`.

```
22 template<typename ...T>
23 promise<T...>::promise();
```

24 Constructs a promise with its results uninitialized and an initial dependency
25 count of 1.

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

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

1 Destroys this promise object.

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

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

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

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

```
8 template<typename ...T>  
9 template<typename ...U>  
10 void promise<T...>::fulfill_result(U &&...results);
```

11 *Precondition:* `fulfill_result` has not been called on this promise before, and
12 the dependency count of this promise is greater than zero.

13 Initializes the promise's result tuple with the given values and decrements the
14 dependency counter by 1. Requires that `T` and `U` have the same number of
15 components, and that each component of `U` is implicitly convertible to the
16 corresponding component of `T`. If the dependency counter reaches zero as a
17 result of this call, the associated future is set to ready, and callbacks that are
18 waiting on the future are executed on the calling thread before this function
19 returns.

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

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

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

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

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

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

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

1 Chapter 6

2 Serialization

3 As a communication library, UPC++ needs to send C++ types between ranks that might
4 be separated by a network interface. The underlying GASNet networking interface sends
5 and receives bytes, thus, UPC++ needs to be able to convert C++ types to and from bytes.

6 6.1 Class Serialization Interface

7 For standard TriviallyCopyable data types, UPC++ can serialize and deserialize these objects
8 for the user without extra intervention on their part. For user data types that have more
9 involved serialization requirements, the user needs to take two steps to inform UPC++ about
10 how to serialize the object.

- 11 1. Declare their type to be a friend of `access`
- 12 2. Implement the visitor function `serialize`

13 The type must also satisfy the C++ CopyConstructible concept.

14 Figure 6.1 provides an example of this process. The definition of the `&` operator for the
15 `Archive` class depends on whether UPC++ is serializing or deserializing an object instance.

16 UPC++ provides implementations of `operator&` for the C++ built-in types. UPC++ se-
17 rialization is compatible with a subset of the Boost serialization interface. This does
18 not imply that UPC++ includes or requires Boost as a dependency. The reference im-
19 plementation of UPC++ does neither of these, it comes with its own implementation of
20 serialization that simply adheres to the interface set by Boost. It is acceptable to have
21 `friend boost::serialization::access` in place of `friend upcxx::access`. UPC++ will
22 use your Boost serialization in that case.

23 There are restrictions on which actions serialization/deserialization routines may per-
24 form. They are:

```

15 class UserType {
16     // The user's fields and member declarations as usual.
17     int member1, member2;
18     // ...
19
20     // To enable the serializer to visit the member fields,
21     // the user provides this...
22     friend class upcxx::access;
23
24     // ...and this
25     template<typename Archive>
26     void serialize(Archive &ar, unsigned) {
27         ar & this->member1;
28         ar & this->member2;
29         // ...
30     }
31 };

```

Figure 6.1: An example of using `access` in a user-defined class

- 1 1. Serialization/deserialization may not call any UPC++ routine with a progress level
2 other than `none`.
- 3 2. UPC++ must perceive these routines as referentially transparent. Loosely, this means
4 that the routines should be “pure” functions between the native representation and
5 a flat sequence of bytes.
- 6 3. The routines must be thread-safe and permit concurrent invocation from multiple
7 threads, even when serializing the same object.

8 6.2 Serialization Concepts

9 UPC++ defines the concepts *DefinitelyTriviallySerializable*, *TriviallySerializable*, *Definitely-*
10 *Serializable*, and *Serializable* that describe what form of serialization a C++ type supports.
11 Figure 6.2 helps summarize the relationship of these concepts.

12 A type `T` is *DefinitelyTriviallySerializable* if either of the following holds:

- 13 • `T` is *TriviallyCopyable* (i.e. `std::is_trivially_copyable<T>::value` is `true`), and
14 if `T` is of class type, `T` does not implement the UPC++ serialization interface described
15 above

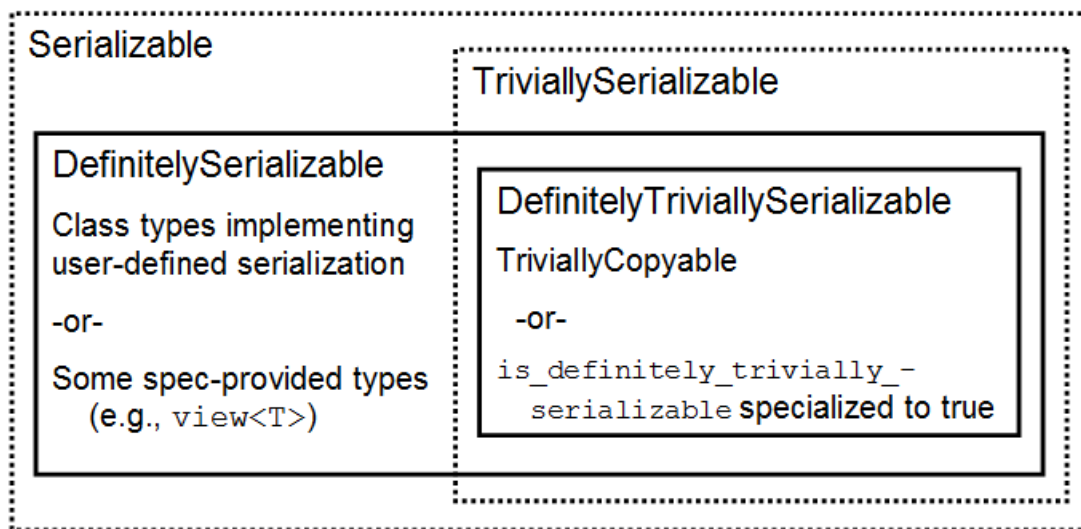


Figure 6.2: Serializable UPC++ concepts type hierarchy. UPC++ concepts with a corresponding trait query are indicated with a solid line.

- 1 • `upcxx::is_definitely_trivially_serializable<T>` is specialized to provide a
2 member constant value that is `true`

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

6 A type `T` is `TriviallySerializable` if it is semantically valid to copy an object by copying
7 its underlying bytes, and UPC++ serializes such types by making a byte copy. A type `T` that
8 is `DefinitelyTriviallySerializable` is also `TriviallySerializable`.

9 A type `T` is `DefinitelySerializable` if one of the following holds:

- 10 • `T` is `DefinitelyTriviallySerializable`
- 11 • `T` is of class type and implements the UPC++ serialization interface
- 12 • `T` is explicitly described as `DefinitelySerializable` by this specification

13 A type `T` is `Serializable` if it is either `TriviallySerializable` or `DefinitelySerializable`.

14 The type trait `upcxx::is_definitely_trivially_serializable<T>` provides a mem-
15 ber constant value that is `true` if `T` is `DefinitelyTriviallySerializable` and `false` otherwise.
16 This trait may be specialized for user types (types that are not defined by the C++ or
17 UPC++ standards).

1 The type trait `upcxx::is_definitely_serializable<T>` provides a member constant
2 value that is `true` if `T` is `DefinitelySerializable` and `false` otherwise. This trait may not
3 be specialized for any types.

4 Several UPC++ communication operations require that the objects to be transferred
5 are of `DefinitelyTriviallySerializable` type. The C++ standard allows implementations to
6 determine whether or not lambda functions are `TriviallyCopyable`, so whether or not such
7 objects are `DefinitelyTriviallySerializable` is implementation-dependent.

8 6.3 Functions

9 In Chapter 7 (*Completion*) and Chapter 9 (*Remote Procedure Calls*) there are several cases
10 where a C++ *FunctionObject* is expected to execute on a destination rank. In these cases
11 the function arguments are serialized as described in this chapter. The *FunctionObject*
12 itself (i.e. the `func` argument to `rpc`, `rpc_ff`, or `as_rpc`) is converted to a function
13 pointer offset from a known *sentinel* in the source program's *code segment*. The details of
14 the implementation are not described here but typical allowed *FunctionObjects* are

- 15 • C functions
- 16 • C++ global and file-scope functions
- 17 • Class static functions
- 18 • lambda functions

19 6.4 Special Handling in Remote Procedure Calls

20 Remote procedure calls, whether standalone (§9) or completion based (§7), perform special
21 handling on certain non-`DefinitelySerializable` UPC++ data structures. Arguments that
22 are either a reference to `dist_object` type (see §13 Distributed Objects) or a `team` (see
23 §12 Teams) are transferred by their `dist_id` or `team_id` respectively. Execution of the
24 RPC is deferred until all of the id's have a corresponding instance constructed on the
25 recipient. When that occurs, `func` is enlisted for execution during user-level progress of
26 the recipient's master persona (see §10 Progress), and it will be called with the recipient's
27 instance references in place of those supplied at the send site. The behavior is undefined if
28 the recipient's instance of a `dist_object` or `team` argument is destroyed before the RPC
29 executes.

6.5 View-Based Serialization

UPC++ also provides a mechanism for serializing the elements of a sequence. The following is an example of transferring a sequence with `rpc`:

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

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 `view` to obtain each element.

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>>`.

UPC++ provides different handling of `view<T>` based on whether the element type `T` is `DefinitelyTriviallySerializable` or not. For `DefinitelyTriviallySerializable` 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 `DefinitelyTriviallySerializable`, 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>`.

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 `DefinitelyTriviallySerializable`, 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.

1 The lifetime of the underlying data buffer and all view iterators on the target in both
 2 the `DefinitelyTriviallySerializable` and `non-DefinitelyTriviallySerializable` cases is restricted
 3 by default to the duration of the RPC. In this case, the elements must be processed or
 4 copied elsewhere before the RPC returns. However, if the RPC returns a future, then the
 5 lifetime of the buffer and view iterators is extended until that future is readied. This allows
 6 an RPC to initiate an asynchronous operation to consume the elements, and as long as the
 7 resulting future is returned from the RPC, the underlying buffer will remain alive until the
 8 asynchronous operation is complete and the future readied.

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

13 The behavior is unspecified when a `view<T, IterType>` is passed to `rpc`, `rpc_ff`, or
 14 `as_rpc` if the type `T` is itself a `view`.

15 6.6 API Reference

```
16 template<typename T>
17 struct is_definitely_trivially_serializable;
```

18 Provides a member constant `value` that is `true` if `T` is `DefinitelyTriviallySeriali-`
 19 `zable` and `false` otherwise. This trait may be specialized for user types.

```
20 template<typename T>
21 struct is_definitely_serializable;
```

22 Provides a member constant `value` that is `true` if `T` is `Definitel-`
 23 `ySerializable` and `false` otherwise. This trait may not be special-
 24 ized. However, its value may be indirectly influenced by specializing
 25 `is_definitely_trivially_serializable<T>` or implementing the class se-
 26 rialization interface for `T`, as appropriate.

```
27 template<typename T>
28 class deserializing_iterator {
29 public:
30 // types
31 using iterator_category = std::input_iterator_tag;
32 using value_type         = T;
33 using difference_type    = std::ptrdiff_t;
```

```
1   using pointer           = T*;
2   using reference         = T;
3
4   deserializing_iterator();
5
6   // accessors
7   T operator*() const;
8
9   // increment
10  deserializing_iterator& operator++();
11  deserializing_iterator operator++(int);
12 };
13
14 // comparisons
15 template<typename T>
16 bool operator==(const deserializing_iterator& x,
17                const deserializing_iterator& y);
18 template<typename T>
19 bool operator!=(const deserializing_iterator& x,
20                const deserializing_iterator& y);
```

21 C++ Concepts: InputIterator

22 An iterator over elements stored in a network buffer. Dereferencing the
23 iterator causes the element to be deserialized and returned by value (i.e.
24 `deserializing_iterator<T>::reference` is an alias for `T`).

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

28 *UPC++ progress level for all functions above: none*

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

31 A type alias that is equivalent to `T*` if `T` is `DefinitelyTriviallySerializable` (i.e.
32 `upcxx::is_definitely_trivially_serializable<T>::value` is true), and
33 `deserializing_iterator<T>` otherwise.

```
34 template<typename T,
35         typename IterType = view_default_iterator_t<T>>
```

```

1 class view {
2 public:
3     // types
4     using iterator    = IterType;
5     using size_type  = std::size_t;
6
7     // iterators
8     iterator begin();
9     iterator end();
10
11    // capacity
12    size_type size() const;
13 };

```

14 C++ Concepts: DefaultConstructible, CopyConstructible, CopyAssignable,
15 Destructible

16 UPC++ Concepts: DefinitelySerializable

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

19 *UPC++ progress level for all member functions of `view`: none*

```

20 template<typename T>
21 class view<T, T*> {
22 public:
23     // types
24     using value_type      = T;
25     using pointer         = T*;
26     using const_pointer   = const T*;
27     using reference       = T&;
28     using const_reference = const T&;
29     using const_iterator  = const T*;
30     using iterator        = const_iterator;
31     using const_reverse_iterator =
32         std::reverse_iterator<const_iterator>;
33     using reverse_iterator = const_reverse_iterator;
34     using size_type        = std::size_t;
35     using difference_type  = std::ptrdiff_t;
36
37     // no explicit construct/copy/destroy for non-owning type
38

```

```
1 // iterators
2 const_iterator      begin() const;
3 const_iterator      cbegin() const;
4
5 const_iterator      end() const;
6 const_iterator      cend() const;
7
8 const_reverse_iterator rbegin() const;
9 const_reverse_iterator crbegin() const;
10
11 const_reverse_iterator rend() const;
12 const_reverse_iterator crend() const;
13
14 // capacity
15 bool empty() const;
16 size_type size() const;
17
18 // element access
19 const_reference operator[](size_type n) const;
20 const_reference at(size_type n) const;
21 const_reference front() const;
22 const_reference back() const;
23
24 const_pointer data() const;
25 };
```

26 C++ Concepts: DefaultConstructible, CopyConstructible, CopyAssignable,
27 Destructible

28 UPC++ Concepts: DefinitelySerializable

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

31 *Exceptions:* `at(n)` throws `std::out_of_range` if `n` is not in the range
32 `[0, size())`.

33 *UPC++ progress level for all member functions of `view`:* none

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

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

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

```
2  template<typename IterType>
3  view<typename std::iterator_traits<IterType>::value_type, IterType>
4    make_view(IterType begin, IterType end,
5              typename std::iterator_traits<IterType>::difference_type
6              size = std::distance(begin, end));
```

7 *Precondition:* `IterType` must satisfy the ForwardIterator C++ concept. The
8 underlying element type (`std::iterator_traits<IterType>::value_type`)
9 must be DefinitelySerializable. `size` must be equal to the number of elements
10 in `[begin, end)`.

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

12 UPC++ progress level: none

```
13  template<typename Container>
14  view<typename Container::value_type,
15       typename Container::const_iterator>
16    make_view(const Container &container);
```

17 *Precondition:* `Container` must satisfy the Container C++ concept. The under-
18 lying element type (`Container::value_type`) must be DefinitelySerializable.

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

21 UPC++ progress level: none

1 Chapter 7

2 Completion

3 7.1 Overview

4 Data movement operations come with the concept of completion, meaning that the effect
5 of the operation is now visible on the source or target rank and that resources, such as
6 memory on the source and destination sides, are no longer in use by UPC++. A single
7 UPC++ call may have several completion events associated with it, indicating completion of
8 different stages of a communication operation. These events are categorized as follows:

- 9 • *Source completion:* The source-side resources of a communication operation are no
10 longer in use by UPC++, and the application is now permitted to modify or reclaim
11 them.
- 12 • *Remote completion:* The data have been deposited on the remote target rank, and
13 they can be consumed by the target.
- 14 • *Operation completion:* The operation is complete from the viewpoint of the initiator.
15 The transferred data can now be read by the initiator, resulting in the values that
16 were written to the target locations.

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

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

- 22 • *Future:* The communication call returns a future, which will be readied when the
23 completion event occurs. This is the default notification mode for communication
24 operations. If the completion event is associated with some values of type $T \dots$, then

1 the returned future will have type `future<T...>`. If no value is associated with the
2 completion, then the future will have type `future<>`.

- 3 • *Promise*: The user provides a promise when requesting notification of a completion
4 event, and that promise will have one its dependencies fulfilled when the event occurs.
5 The promise must have a non-zero dependency count. If the completion event is asso-
6 ciated with some values of type `T...`, then it must be valid to call `fulfill_result()`
7 on the promise with values of type `T...`, and the promise must not have had
8 `fulfill_result()` called on it. The promise will then have `fulfill_result()`
9 called on it with the associated values when the completion event occurs. If no value
10 is associated with the completion, then the promise may have any type. It will have
11 an anonymous dependency fulfilled upon the completion event.
- 12 • *Local-Procedure Call (LPC)*: The user provides a target persona and a callback func-
13 tion object when requesting notification of a completion event. If the completion
14 is associated with some values of type `T...`, then the callback must be invocable
15 with arguments of type `T...`. Otherwise, it must be invocable with no arguments.
16 The callback, together with the associated completion values if any, is enlisted for
17 execution on the given persona when the completion event occurs.
- 18 • *Remote-Procedure Call (RPC)*: The user provides a Serializable function object when
19 requesting notification of a completion event, as well as the arguments on which the
20 function object should be invoked. Each argument must either be DefinitelySeri-
21 alizable, a `dist_object<T>&`, or `team&`. The function object and arguments are
22 transferred as part of the communication operation, and the invocation is enlisted
23 for execution on the master persona of the target rank when the completion event
24 occurs.
- 25 • *Buffered*: The communication call consumes the source-side resources of the operation
26 before the call returns, allowing the application to immediately modify or reclaim
27 them. This delays the return of the call until after the source-completion event. The
28 implementation may internally buffer the source-side resources or block until network
29 resources are available to inject the data directly.
- 30 • *Blocking*: This is similar to buffered completion, except that the implementation is
31 required to block until network resources are available to inject the data directly.

32 Future, promise, and LPC completions are only valid for completion events that occur
33 at the initiator of a communication call, namely source and operation completion. RPC
34 completion is only valid for a completion event that occurs at the target of a communication
35 operation, namely remote completion. Buffered and blocking completion are only valid for

1 source completion. More details on futures and promises are in Ch. 5, while LPC and
2 RPC callbacks are discussed in Ch. 10.

3 Notification of completion only happens during user-level progress of the initiator or
4 target rank. Even if an operation completes early, including before the initiation operation
5 returns, the application cannot learn this fact without entering user progress. For futures
6 and promises, only when the initiating thread (persona actually) enters user-level progress
7 will the future or promise be eligible for taking on a readied or fulfilled state. LPC callbacks
8 will execute once a thread enters user progress of the designated persona. See Ch. 10 for
9 the full discussion on user progress and personas.

10 If buffered or blocking completion is requested, then the source-completion event occurs
11 before the communication call returns. However, source-completion notifications, such as
12 triggering a future or executing an LPC, are still delayed until the next user-level progress.

13 Operation completion implies both source and remote completion. However, it does not
14 imply that the actions associated with source and remote completion have been executed.

15 7.2 Completion Objects

16 The UPC++ mechanism for requesting notification of completion is through opaque *com-*
17 *pletion objects*, which associate notification actions with completion events. Completion
18 objects are CopyConstructible, CopyAssignable, and Destructible, and the same comple-
19 tion object may be passed to multiple communication calls. A simple completion object
20 is constructed by a call to a static member function of the `source_cx`, `remote_cx`, or
21 `operation_cx` class, providing notification for the corresponding event. The member func-
22 tions `as_future`, `as_promise`, `as_lpc`, and `as_rpc` request notification through a future,
23 promise, LPC, or RPC, respectively. Only the member functions that correspond to valid
24 means of signaling notification of an event are defined in the class associated with that
25 event.

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

```
27 global_ptr<int> gp1 = /* some global pointer */;  
28 promise<int> pro1;  
29 auto cxs = operation_cx::as_promise(pro1);  
30 rget(gp1, cxs);  
31 pro1.finalize(); // fulfill the initial anonymous dependency
```

32 The `rget` function, when provided just a `global_ptr<int>`, transfers a single `int` from
33 the given location to the initiator. Thus, operation completion is associated with an `int`
34 value, and the promise used for signaling that event must have type compatible with an
35 `int` value, e.g. `promise<int>`. The user constructs a completion object that requests
36 operation notification on the promise `pro1` by calling `operation_cx::as_promise(pro1)`.
37 Since a completion object is opaque, the `auto` keyword is used to deduce the type of the

1 completion object. The resulting completion object can then be passed to `rget`, which
 2 fulfills the promise with the transferred value upon operation completion.

3 A user can request notification of multiple completion events, as well as multiple no-
 4 tifications of a single completion event. The pipe (`|`) operator can be used to combine
 5 completion objects to construct a union of the operands. The following is an example:

```

6 int foo() {
7     return 0;
8 }
9
10 int bar(int x) {
11     return x;
12 }
13
14 void do_comm(double *src, size_t count) {
15     global_ptr<double> dest = /* some global pointer */;
16     promise<> pro1;
17     persona &per1 = /* some persona */;
18     auto cxs = (operation_cx::as_promise(pro1) |
19               source_cx::as_future() |
20               operation_cx::as_future() |
21               operation_cx::as_future() |
22               source_cx::as_lpc(per1, foo) |
23               remote_cx::as_rpc(bar, 3)
24               );
25     std::tuple<future<>, future<>, future<>> result =
26         rput(src, dest, count, cxs);
27     pro1.finalize().wait(); // finalize promise, wait on its future
28 }

```

29 This code initiates an `rput` operation, which provides source-, remote-, and operation-
 30 completion events. A unified completion object is constructed by applying the pipe op-
 31 erator to individual completion objects. When `rput` is invoked with the resulting unified
 32 completion object, it returns a tuple of futures corresponding to the individual future com-
 33 pletions requested. The ordering of futures in this tuple matches the order of application
 34 of the pipe operator (this operator is associative but not commutative). In the example
 35 above, the first future in the tuple would correspond to source completion, and the second
 36 and third would be for operation completion. If no future-based notification is requested,
 37 then the return type of the communication call would be `void` rather than a tuple.

38 When multiple notifications are requested for a single event, the order in which those
 39 notifications occur is unspecified. In the code above, the order in which `pro1` is fulfilled
 40 and the two futures for operation completion are readied is indeterminate. Similarly, if

1 both source and operation completion occur before the next user-level progress, the order
2 in which the notifications occur is unspecified, so that operation-completion requests may
3 be notified before source-completion requests.

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

7 Arguments to `remote_cx::as_rpc` are serialized at an unspecified time between the
8 invocation of `as_rpc` and the source completion event of a communication operation that
9 accepts the resulting completion object. If multiple communication operations use a single
10 completion object resulting from `as_rpc`, then the arguments may be serialized multiple
11 times. For arguments that are not passed by value, the user must ensure that they re-
12 main valid until source completion of all communication operations that use the associated
13 completion object.

14 7.2.1 Restrictions

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

21 If a UPC++ call provides both operation and remote completion, then at least one must
22 be requested by the provided completion object. If a call provides operation but not remote
23 completion, then operation completion must be requested. The behavior of the program is
24 undefined if neither operation nor remote completion is requested from a call that supports
25 one or both of operation or remote completion.

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

30 7.2.2 Completion and Return Types

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

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

- 1 • `future<T...>`, if a single future-based completion is requested, where `T...` is the
 - 2 sequence of types associated with the given completion event
 - 3 • `std::tuple<future<T...>...>`, if multiple future-based completions are requested,
 - 4 where each future's arguments `T...` is the sequence of types associated with the
 - 5 corresponding completion event
- 6 Type deduction, such as with `auto`, is recommended when working with completion objects
- 7 and return types.

8 7.2.3 Default Completions

9 If a completion object is not explicitly provided to a communication call, then a default

10 completion object is used. For most calls, the default is `operation_cx::as_future()`.

11 However, for `rpc_ff`, the default completion is `source_cx::as_buffered()`, and for `rpc`,

12 it is `source_cx::as_buffered() | operation_cx::as_future()`. The default comple-

13 tion of a UPC++ communication call is listed in its API reference.

14 7.3 API Reference

```
15 struct source_cx;
```

```
16
```

```
17 struct remote_cx;
```

```
18
```

```
19 struct operation_cx;
```

20 Types that contain static member functions for constructing completion objects

21 for source, remote, and operation completion.

```
22 [static] CType source_cx::as_future();
```

```
23
```

```
24 [static] CType operation_cx::as_future();
```

25 Constructs a completion object that represents notification of source or opera-

26 tion completion with a future.

27 *UPC++ progress level: none*

```
28 template<typename ...T>
```

```
29 [static] CType source_cx::as_promise(promise<T...> &pro);
```

```
30
```

```
31 template<typename ...T>
```

```
32 [static] CType operation_cx::as_promise(promise<T...> &pro);
```

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

2 Constructs a completion object that represents signaling the given promise
3 upon source or operation completion.

4 *UPC++ progress level:* `none`

```
5  template<typename Func>  
6  [static] CType source_cx::as_lpc(persona &target, Func func);
```

```
7  
8  template<typename Func>  
9  [static] CType operation_cx::as_lpc(persona &target, Func func);
```

10 *Preconditions:* `Func` must be a function-object type and `CopyConstructible`.
11 `func` must not throw an exception when invoked.

12 Constructs a completion object that represents the enqueueing of `func` on the
13 given local `persona` upon source or operation completion.

14 *UPC++ progress level:* `none`

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

17 *Precondition:* `Func` must be `Serializable` and `CopyConstructible` and a function-
18 object type. Each of `Args...` must either be a `DefinitelySerializable` and `Copy-`
19 `Constructible` type, or `dist_object<T>&`, or `team&`. The call `func(args...)`
20 must not throw an exception.

21 Constructs a completion object that represents the enqueueing of `func` on a
22 target rank upon remote completion.

23 *UPC++ progress level:* `none`

```
24  [static] CType source_cx::as_buffered();
```

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

28 *UPC++ progress level:* `none`

```
29  [static] CType source_cx::as_blocking();
```

1 Constructs a completion object that represents blocking until source-side re-
2 sources are consumed before a communication call returns, delaying the return
3 until the source-completion event occurs.

4 *UPC++ progress level: none*

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

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

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

11 *UPC++ progress level: none*

1 Chapter 8

2 One-Sided Communication

3 8.1 Overview

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

8 8.2 API Reference

9 8.2.1 Remote Puts

```
10 template<typename T,  
11         typename Completions=decltype(operation_cx::as_future())>  
12 RType rput(T value, global_ptr<T> dest,  
13           Completions cxs=Completions{});
```

14 *Precondition:* `T` must be **DefinitelyTriviallySerializable**. `dest` must reference a
15 valid object of type `T`.

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

17 *Completions:*

- 18 • *Remote:* Indicates completion of the transfer of `value`.
- 19 • *Operation:* Indicates completion of all aspects of the operation: the trans-
20 fer and remote stores are complete.

21 *C++ memory ordering:* The writes to `dest` will have a *happens-before* re-
22 lationship 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.

UPC++ progress level: internal

```

6  template<typename T,
7      typename Completions=decltype(operation_cx::as_future())>
8  RType rput(T const *src, global_ptr<T> dest, std::size_t count,
9      Completions cxs=Completions{});

```

Precondition: T must be DefinitelyTriviallySerializable. Addresses `src+i` and `dst+i` for `i` in `[0, count)` must all reference valid and non-overlapping objects of type T.

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.

Completions:

- *Source:* Indicates completion of injection or internal buffering of the source values, signifying that the `src` buffer may be modified.
- *Remote:* Indicates completion of the transfer of the values, implying readiness of the target buffer `[dest, dest+count)`.
- *Operation:* Indicates completion of all aspects of the operation: the transfer and remote stores are complete.

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.

UPC++ progress level: internal

1 8.2.2 Remote Gets

```
2  template<typename T,  
3      typename Completions=decltype(operation_cx::as_future())>  
4  RType rget(global_ptr<T> src,  
5      Completions cxs=Completions{});
```

6 *Precondition:* T must be DefinitelyTriviallySerializable. `src` must reference a
7 valid object of type T.

8 Initiates a transfer to this rank of a single value of type T located at `src`. The
9 value will be transferred to the calling rank and delivered in the operation-
10 completion notification.

11 *Completions:*

- 12 • *Operation:* Indicates completion of all aspects of the operation, including
13 transfer and readiness of the resulting value. This completion produces a
14 value of type T.

15 *C++ memory ordering:* The read of `src` will have a *happens-before* relationship
16 with the operation-completion notification actions (future readying, promise
17 fulfillment, or persona LPC enlistment). All evaluations *sequenced-before* this
18 call will have a *happens-before* relationship with the invocation of any LPC
19 associated with operation completion.

20 *UPC++ progress level:* **internal**

```
21  template<typename T,  
22      typename Completions=decltype(operation_cx::as_future())>  
23  RType rget(global_ptr<T> src, T *dest, std::size_t count,  
24      Completions cxs=Completions{});
```

25 *Precondition:* T must be DefinitelyTriviallySerializable. Addresses `src+i` and
26 `dst+i` for `i` in `[0, count)` must all reference valid and non-overlapping objects
27 of type T.

28 Initiates a transfer of `count` values of type T beginning at `src` and stores them
29 in the locations beginning at `dest`. The source values must not be modified
30 until operation completion is notified.

31 *Completions:*

- 32 • *Operation:* Indicates completion of all aspects of the operation, including
33 transfer and readiness of the resulting values.

1 *C++ memory ordering:* The reads of `src` and writes to `dest` will have a
2 *happens-before* relationship with the operation-completion notification actions
3 (future readying, promise fulfillment, or persona LPC enlistment). All evalua-
4 tions *sequenced-before* this call will have a *happens-before* relationship with the
5 invocation of any LPC associated with operation completion.
6 *UPC++ progress level:* `internal`

1 Chapter 9

2 Remote Procedure Call

3 9.1 Overview

4 UPC++ provides remote procedure calls (RPCs) for injecting function calls into other ranks.
5 These injections are one-sided, meaning the recipient is not required to explicitly acknowl-
6 edge which functions are expected. Concurrent with a rank's execution, incoming RPCs
7 accumulate in an internal queue managed by UPC++. The only control a rank has over
8 inbound RPCs is when it would like to check its inbox for arrived function calls and execute
9 them. Draining the RPC inbox is one of the many responsibilities of the progress API (see
10 Ch. 10, *Progress*).

11 There are two main flavors of RPC in UPC++: *fire-and-forget* (`rpc_ff`) and *round trip*
12 (`rpc`). Each takes a function `Func` together with variadic arguments `Args`.

13 The `rpc_ff` call serializes the given function and arguments into a message destined
14 for the recipient, and guarantees that this function call will be placed eventually in the
15 recipient's inbox. The round-trip `rpc` call does the same, but also forces the recipient to
16 reply to the sender of the RPC with a message containing the return value of the function,
17 providing the value for operation completion of the sender's invocation of `rpc`. Thus,
18 when the future is ready, the sender knows the recipient has executed the function call.
19 Additionally, if the return value of `func` is a future, the recipient will wait for that future
20 to become ready before sending its result back to the sender.

21 There are important restrictions on what the permissible types for `func` and its bound
22 arguments can be for RPC functions. First, the `Func` type must be a function object (has
23 a publicly accessible overload of the function call operator, `operator()`). Second, `Func`
24 must be `Serializable`, and all `Args...` types must be `DefinitelySerializable` (see Ch. 6,
25 *Serialization*).

9.2 Remote Hello World Example

Figure 9.1 shows a simple alternative *Hello World* example where each rank issues an `rpc` to its neighbor, where the last rank wraps around to 0.

```

32 #include <upcxx/upcxx.hpp>
33 #include <iostream>
34 void hello_world(intrank_t num){
35     std::cout << "Rank " << num <<"   told rank " << upcxx::rank_me()
36         << " to say Hello World" << std::endl;
37 }
38 int main(int argc, char** argv){
39     upcxx::init();           // Start UPC++ state
40     intrank_t remote = (upcxx::rank_me()+1)%upcxx::rank_n();
41     auto f = upcxx::rpc(remote, hello_world, upcxx::rank_me());
42     f.wait();
43     upcxx::finalize();     // Close down UPC++ state
44     return 0;
45 }

```

Figure 9.1: HelloWorld with Remote Procedure Call

9.3 API Reference

```

5 template<typename Func, typename ...Args>
6 void rpc_ff(intrank_t recipient, Func &&func, Args &&...args);
7 template<typename Completions, typename Func, typename ...Args>
8 RType rpc_ff(intrank_t recipient, Completions cxs,
9             Func &&func, Args &&...args);

```

Precondition: `Func` must be a Serializable type and a function-object type. Each of `Args...` must be a DefinitelySerializable type, or `dist_object<T>&`, or `team&`. The call `func(args...)` must not throw an exception.

In the first variant, the `func` and `args...` are serialized and internally buffered before the call returns. The call `rpc_ff(rank, func, args...)` is equivalent to `rpc_ff(rank, source_cx::as_buffered(), func, args...)`.

In the second variant, if buffered source completion is not requested, the `func` and `args...` are serialized at an unspecified time between the invocation of

1 `rpc_ff` and source completion. The serialized results are retained internally
2 until they are eventually sent.

3 After their receipt on `recipient`, the data are deserialized and `func(args...)`
4 is enlisted for execution during user-level progress of the master persona. So
5 long as the sending persona continues to make internal-level progress it is guar-
6 anteed that the message will eventually arrive at the recipient. See §10.5.3
7 `progress_required` for an understanding of how much internal-progress is
8 necessary.

9 The execution of `func(args...)` is never performed synchronously, even if
10 the target is the same as the calling rank and this function is invoked during
11 user-level progress.

12 Special handling is applied to those members of `args` which are either a refer-
13 ence to `dist_object` type or a `team`, as described in §6.4.

14 *Completions:*

- 15 • *Source:* Indicates completion of serialization of the function object and
16 arguments.

17 *C++ memory ordering:* All evaluations *sequenced-before* this call will have
18 a *happens-before* relationship with the source-completion notification actions
19 (future readying, promise fulfillment, or persona LPC enlistment) and the re-
20 cipient's invocation of `func`.

21 *UPC++ progress level:* `internal`

```
22  template<typename Func, typename ...Args>  
23  future_invoke_result_t<Func, Args...>  
24  rpc(intrank_t recipient, Func &&func, Args &&...args);  
25  template<typename Completions, typename Func, typename ...Args>  
26  RType rpc(intrank_t recipient, Completions cxs,  
27           Func &&func, Args &&...args);
```

28 *Precondition:* `Func` must be a `Serializable` type and a function-
29 object type. Each of `Args...` must be either a `DefinitelySe-`
30 `rializable` type, or `dist_object<T>&`, or `team&`. Additionally,
31 `std::result_of<Func(Args...)>::type` must be either a `DefinitelySe-`
32 `rializable` type that is not `view<U, IterType>`, or `future<T...>`, where each
33 type in `T...` must be `DefinitelySerializable` but not `view<U, IterType>`. The
34 call `func(args...)` must not throw an exception.

1 Similar to `rpc_ff`, this call sends `func` and `args...` to be executed remotely,
2 but additionally provides an operation-completion event that produces the
3 value returned from the remote invocation of `func(args...)`, if it is non-void.

4 In the first variant, the `func` and `args...` are serialized and internally buffered
5 before the call returns. The call `rpc(rank, func, args...)` is equivalent to

```
6 rpc(rank ,  
7     source_cx::as_buffered() | operation_cx::as_future(),  
8     func, args...)
```

9 In the second variant, if buffered source completion is not requested, the `func`
10 and `args...` are serialized at an unspecified time between the invocation of
11 `rpc` and source completion. The serialized results are retained internally until
12 they are eventually sent.

13 After their receipt on `recipient`, the data are deserialized and `func(args...)`
14 is enlisted for execution during user-level progress of the master persona.

15 In the first variant, the returned future is readied upon operation completion.

16 For futures provided by an operation-completion request, or promises used in
17 promise-based operation-completion requests, the type of the future or promise
18 must correspond to the return type of `func(args...)` as follows:

- 19 • If the return type is of the form `future<T...>`, then a future provided by
20 operation completion also has type `future<T...>`, and promises used in
21 operation-completion requests must permit invocation of `fulfill_result`
22 with values of type `T...`
- 23 • If the return type is some other non-void type `T`, then a future provided by
24 operation completion has type `future<T>`, and promises used in operation-
25 completion requests must permit invocation of `fulfill_result` with a
26 value of type `T`.
- 27 • If the return type is `void`, then a future provided by operation completion
28 has type `future<>`, and promises used in operation-completion requests
29 may have any type `promise<T...>`.

30 Within user-progress of the recipient's master persona, the result from invoking
31 `func(args...)` will be immediately serialized and eventually sent back to the
32 initiating rank. Upon receipt, it will be deserialized, and operation-completion
33 notifications will take place during subsequent user-progress of the initiating
34 persona.

1 The execution of `func(args...)` is never performed synchronously, even if
2 the target is the same as the calling rank and this function is invoked during
3 user-level progress.

4 The same special handling applied to `dist_object&` and `team&` arguments by
5 `rpc_ff` is also done by `rpc`.

6 *Completions:*

- 7 • *Source:* Indicates completion of serialization of the function object and
8 arguments.
- 9 • *Operation:* Indicates completion of all aspects of the operation: serial-
10 ization, deserialization, remote invocation, transfer of any result, and de-
11 struction of any internally managed values are complete. This completion
12 produces a value as described above.

13 *C++ memory ordering:* All evaluations *sequenced-before* this call will have a
14 *happens-before* relationship with the invocation of `func`. The return from `func`,
15 will have a *happens-before* relationship with the operation-completion actions
16 (future readying, promise fulfillment, or persona LPC enlistment). For LPC
17 completions, all evaluations *sequenced-before* this call will have a *happens-before*
18 relationship with the execution of the completion function.

19 *UPC++ progress level:* **internal**

1 Chapter 10

2 Progress

3 10.1 Overview

4 UPC++ presents a highly-asynchronous interface, but guarantees that user-provided call-
5 backs will only ever run on user threads during calls to the library. This guarantees a good
6 user-visibility of the resource requirements of UPC++, while providing a better interoper-
7 ability with other software packages which may have restrictive threading requirements.
8 However, such a design choice requires the application developer to be conscientious about
9 providing UPC++ access to CPU cycles.

10 Progress in UPC++ refers to how the calling application allows the UPC++ internal run-
11 time to advance the state of its outstanding asynchronous operations. Any asynchronous
12 operation initiated by the user may require the application to give UPC++ access to the exe-
13 cution thread periodically until the operation reports its completion. Such access is granted
14 by simply making calls into UPC++. Each UPC++ function’s contract to the user contains its
15 *progress guarantee* level. This is described by the members of the `upcxx::progress_level`
16 enumerated type:

17 `progress_level::user` UPC++ may advance its internal state as well as signal completion
18 of user-initiated operations. This may entail the firing of remotely injected procedure
19 calls (RPCs), or readying/fulfillment of futures/promises and the ensuing callback
20 cascade.

21 `progress_level::internal` UPC++ may advance its internal state, but no notifications
22 will be delivered to the application. Thus, an application has very limited ways to
23 “observe” the effects of such progress.

24 *Progress level: none* UPC++ will not attempt to advance the progress of asynchronous op-
25 erations. (Note this level does not have an explicit entry in the `progress_level`
26 enumerated type).

1 The most common progress guarantee made by UPC++ functions is `progress_level::`
2 `internal`. This ensures the delivery of notifications to remote ranks (or other threads)
3 making *user*-level progress in a timely manner. In order to avoid having the user contend
4 with the cost associated with callbacks and RPCs being run anytime a UPC++ function is
5 entered, `progress_level::user` is purposefully not the common case.

6 `progress` is the notable function enabling the application to make *user*-level progress.
7 Its sole purpose is to look for ready operations involving this rank or thread and run the
8 associated RPC/callback code.

```
9 upcxx::progress(progress_level lev = progress_level::user)
```

10 UPC++ execution phases which leverage asynchrony heavily tend to follow a particular
11 program structure. First, initial communications are launched. Their completion callbacks
12 might then perform a mixture of compute or further UPC++ communication with similar,
13 cascading completion callbacks. Then, the application spins on `upcxx::progress()`,
14 checking some designated application state which monitors the amount of pending outgoing/
15 incoming/local work to be done. For the user, understanding which functions perform
16 these progress spins becomes crucial, since any invocation of user-level progress may execute
17 RPCs or callbacks.

18 10.2 Restricted Context

19 During user-level progress made by UPC++, callbacks may be executed. Such callbacks
20 are subject to restrictions on how they may further invoke UPC++ themselves. We designate
21 such restricted execution of callbacks as being in the *restricted context*. The general
22 restriction is stated as:

23 *User code running in the restricted context must assume that for the duration*
24 *of the context all other attempts at making user-level progress, from any thread*
25 *on any rank, may result in a no-op every time.*

26 The immediate implication is that a thread which is already in the restricted context
27 should assume no-op behavior from further attempts at making progress. This makes it
28 pointless to try and wait for UPC++ notifications from within restricted context since there
29 is no viable mechanism to make the notifications visible to the user. Thus, calling any
30 routine which spins on user-level progress until some notification occurs will likely hang
31 the thread.

10.3 Attentiveness

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 rank 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.

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 all operations initiated by this rank have reached a state at which they no longer require progress. This is achieved by using the following functions:

```
bool upcxx::progress_required();
void upcxx::discharge();
```

The `progress_required` function reports whether this rank requires progress, allowing the application to know that there are still pending operations that will not achieve remote completion without further advancements to internal progress. This is of particular importance before an application enters a lapse of inattentiveness (for instance, performing expensive computations) in order to prevent slowing down remote entities.

The `discharge` function allows an application to ensure that UPC++ does not require progress anymore. It is equivalent to the following:

```
void upcxx::discharge() {
    while(upcxx::progress_required())
        upcxx::progress(upcxx::progress_level::internal);
}
```

A well-behaved UPC++ application is encouraged to call `discharge` before any long lapse of attentiveness to progress.

10.4 Thread Personas/Notification Affinity

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

1 about how a multi-threaded application can safely use futures returned by UPC++ calls.

2 The most important thing an application has to be aware of is which thread UPC++
3 will use to signal completion of a given future. It is therefore extremely important to
4 know that UPC++ will use the same thread to which the future was returned by the UPC++
5 operation (i.e. the thread which invoked the operation in the first place). This means
6 that the thread which invoked a future-returning operation will be the only one able to
7 see that operation's completion. As UPC++ triggers futures only during a call which makes
8 user-level progress, the invoking thread must continue to make such progress calls until
9 the future is satisfied. This requirement has the drawback of banning the application from
10 doing the following: initiating a future-returning operation on one thread, allowing that
11 thread to terminate or become permanently inattentive (e.g. sleeping in a thread pool),
12 and expecting a different thread to receive the future's completion. This section will focus
13 on two ways the application can still attain this use-case.

14 The notion of “thread” has been used in a loose fashion throughout this document,
15 the natural interpretation being an operating system (OS) thread. More precisely, this
16 document uses the notion of “thread” to denote a UPC++ device referred to as *thread persona*
17 which generalizes the notion of operating system threads.

18 A UPC++ thread persona is a collection of UPC++-internal state usually attributed to a
19 single thread. By making it a proper construct, UPC++ allows a single OS thread to switch
20 between multiple application-defined roles for processing notifications. Personas act as the
21 receivers for notifications generated by the UPC++ runtime.

22 Values of type `upcxx::persona` are non-copyable, non-moveable objects which the
23 application can instantiate as desired. For each OS thread, UPC++ internally maintains
24 a stack of *active* persona references. The top of this stack is the *current* persona. All
25 asynchronous UPC++ operations will have their notification events (signaling of futures or
26 promises) sent to the current persona of the OS thread invoking the operation. Calls that
27 make user-level progress will process notifications destined to any of the active personas of
28 the invoking thread. The initial state of the persona stack consists of a single entry pointing
29 to a persona created by UPC++ which is dedicated to the current OS thread. Therefore,
30 if the application never makes any use of the persona API, notifications will be processed
31 solely by the OS thread that initiates the operation.

32 Pushing and popping personas from the persona stack (hence changing the current
33 persona) is done with the `upcxx::persona_scope` type.

```
34 namespace upcxx {  
35  
36     struct persona_scope {  
37         // Make 'p' the new current persona for this OS thread.  
38         persona_scope(persona &p);  
39  
40         // Acquire 'lock', then make 'p' the new current persona for
```

```

1     // this OS thread.
2     template<typename Lock>
3     persona_scope(Lock &lock, persona &p);
4
5     // Pop 'p' from persona stack, release 'lock' if any.
6     // Calling thread must be same for constructor and destructor.
7     ~persona_scope();
8 };
9
10    persona_scope& top_persona_scope();
11
12    persona_scope& default_persona_scope();
13
14    bool progress_required(persona_scope &ps = top_persona_scope());
15
16    void discharge(persona_scope &ps = top_persona_scope());
17
18 } // namespace upcxx
19
20 // Example demonstrating persona_scope.
21
22 persona scheduler_persona;
23 std::mutex scheduler_lock;
24
25 { // Scope block delimits domain of persona_scope instance.
26     auto scope = persona_scope(scheduler_lock, scheduler_persona);
27
28     // All following upcxx actions will use 'scheduler_persona'
29     // as current.
30
31     // ...
32
33     // 'scope' destructs:
34     // - 'scheduler_persona' dropped from active set if it
35     //   wasn't active before the scope's construction.
36     // - Previously current persona revived.
37     // - Lock released.
38 }

```

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.

1 There are two ways that asynchronous operations can be initiated by a given OS thread
2 but retired in another. The first solution is simple:

- 3 1. The user defines a persona P.
- 4 2. Thread 1 activates P, initiates the asynchronous operation, and releases P.
- 5 3. Thread 1 synchronizes with Thread 2, indicating the operation has been initiated.
- 6 4. Thread 2 activates P, spins on **progress** until the operation completes.

7 Care must be taken that any futures created by phase 2 are never altered (uttered)
8 concurrently. The same synchronization that was used to enforce exclusivity of persona
9 acquisition can be leveraged to protect the future as well.

10 While this technique achieves our goal of different threads initiating and resolving
11 asynchronous operations, it fails a different but also desirable property. It is often desirable
12 to allow multiple threads to issue communication *concurrently* while delegating a separate
13 thread to handle the notifications. To achieve this, it is clear that multiple personas are
14 needed. Indeed, the exclusivity of a persona being current to only one OS thread prevents
15 the application from concurrent initiation of communication.

16 In order to issue operations and concurrently retire them in a different thread, the user
17 is strongly encouraged to use the LPC completion mechanism described in Chapter 7, as
18 opposed to the future or promise variants. An example of such a call is:

```
19 rget(gp_ptr_src, operation_cx::as_lpc(some_persona, callback_func));
```

20 In addition to the arguments necessary for the particular operation, the `as_lpc` com-
21 pletion mechanism takes a persona reference and a C++ function object (lambda, etc.) such
22 that upon completion of the operation, the designated persona shall execute the function
23 object during its user-level progress. Using this mechanism, it is simple to have multi-
24 ple threads initiating communication concurrently with a designated thread receiving the
25 completion notifications. To achieve this, each operation is initiated by a thread using the
26 agreed-upon persona of the receiver thread together with a callback that will incorporate
27 knowledge of completion into the receiver's state.

28 10.5 API Reference

```
29 enum class progress_level {  
30     /*none, -- not an actual member, conceptual only*/  
31     internal,  
32     user  
33 };
```

```
1 void progress(progress_level lev = progress_level::user);
```

2 This call will always attempt to advance internal progress.

3 If `lev == progress_level::user` then this thread is also used to execute any
4 available user actions for the personas currently active. Actions include:

- 5 1. Either future-readying or promise-fulfilling completion notifications for
6 asynchronous operations initiated by one of the active personas. By the
7 execution model of futures and promises this can induce callback cascade.
- 8 2. Continuation-style completion notifications from operations initiated by
9 any persona but designating one of the active personas as the completion
10 recipient.
- 11 3. RPCs destined for this rank but only if the master persona is among the
12 active set.
- 13 4. `lpc`'s destined for any of the active personas.

14 *UPC++ progress level: internal or user*

15 10.5.1 persona

```
16 class persona;
```

17 C++ Concepts: DefaultConstructible, Destructible

```
18 persona::persona();
```

19 Constructs a persona object with no enqueued operations.

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

21 *UPC++ progress level: none*

```
22 persona::~~persona();
```

23 Destructs this persona object. If this persona is a member of any thread's
24 persona stack, the result of this call is undefined. If any operations are currently
25 enqueued on this persona, or if any operations initiated by this persona require
26 further progress, the result of this call is undefined.

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

28 *UPC++ progress level: none*

```
1  template<typename Func>
2  void persona::lpc_ff(Func func);
```

3 *Precondition:* `Func` must be a function-object type that can be invoked on zero
4 arguments, and the call `func()` must not throw an exception.

5 `std::move`'s `func` into an unordered collection of type-erased function objects
6 to be executed during user-level progress of the targeted (this) persona. This
7 function is thread-safe, so it may be called from any thread to enqueue work
8 for this persona.

9 The execution of `func` is never performed synchronously, even if the target
10 persona is a member of the caller's persona stack and this function is invoked
11 during user-level progress.

12 *C++ memory ordering:* All evaluations *sequenced-before* this call will have a
13 *happens-before* relationship with the invocation of `func`.

14 *UPC++ progress level:* none

```
15  template<typename Func>
16  future_invoke_result_t<Func> persona::lpc(Func func);
```

17 *Precondition:* `Func` must be a function-object type that can be invoked on zero
18 arguments, and the call `func()` must not throw an exception.

19 `std::move`'s `func` into an unordered collection of type-erased function objects
20 to be executed during user-level progress of the targeted (this) persona. The
21 return value of `func` is asynchronously returned to the currently active persona
22 in a future. If the return value of `func` is a future, then the targeted persona will
23 wait for that future before signaling the future returned by `lpc` with its value.
24 This function is thread-safe, so it may be called from any thread to enqueue
25 work for this persona. Note that the future returned by `lpc` is considered to
26 be owned by the currently active persona, the future returned by `func` (if any)
27 will be considered owned by the target (this) persona.

28 The execution of `func` is never performed synchronously, even if the target
29 persona is a member of the caller's persona stack and this function is invoked
30 during user-level progress.

31 *C++ memory ordering:* All evaluations *sequenced-before* this call will have a
32 *happens-before* relationship with the invocation of `func`, and the invocation of
33 `func` will have a *happens-before* relationship with evaluations sequenced after
34 the signaling of the final future.

35 *UPC++ progress level:* none

```
1 persona& master_persona();
```

2 Returns a reference to the master persona automatically instantiated by the
3 UPC++ runtime. The thread that executes `upcxx::init` implicitly acquires this
4 persona as its current persona. The master persona is special in that it is the
5 only one which will execute RPCs destined for this rank. Additionally, some
6 UPC++ functions may only be called by a thread with the master persona in its
7 active stack.

8 *UPC++ progress level: none*

```
9 persona& current_persona();
```

10 Returns a reference to the persona on the top of the thread's active persona
11 stack.

12 *UPC++ progress level: none*

```
13 persona& default_persona();
```

14 Returns a reference to the persona instantiated automatically and uniquely for
15 this OS thread. The default persona is always the bottom of and can never be
16 removed from its designated OS thread's active stack.

17 *UPC++ progress level: none*

```
18 void liberate_master_persona()
```

19 *Precondition:* This thread must be the one which called `upcxx::init`, it must
20 have not altered its persona stack since calling `init`, and it must not have
21 called this function already since calling `init`.

22 The thread which invokes `upcxx::init` implicitly has the master persona at
23 the top of its active stack, yet the user has no `persona_scope` to drop to allow
24 other threads to acquire the persona. Thus, if the user intends for other threads
25 to acquire the master persona, they should have the `init`-calling thread release
26 the persona with this function so that it can be claimed by `persona_scope`'s.
27 Generally, if this function is ever called, it is done soon after `init` and then the
28 master persona should be reacquired by a `persona_scope`.

29 *UPC++ progress level: none*

1 10.5.2 `persona_scope`

2 `class persona_scope;`

3 C++ Concepts: Destructible, MoveConstructible

4 `persona_scope::persona_scope(persona &p);`

5 *Precondition:* Excluding this thread, `p` is not a member of any other thread's
6 active stack.

7 Pushes `p` onto the top of the calling OS thread's active persona stack.

8 *UPC++ progress level:* none

9 `template<typename Mutex>`

10 `persona_scope::persona_scope(Mutex &mutex, persona &p);`

11 C++ Concepts of `Mutex`: `Mutex`

12 *Precondition:* `p` will only be a member of some thread's active stack if that
13 thread holds `mutex` in a locked state.

14 Invokes `mutex.lock()`, then pushes `p` onto the OS thread's active persona
15 stack.

16 *UPC++ progress level:* none

17 `persona_scope::~~persona_scope();`

18 *Precondition:* All `persona_scope`'s constructed on this thread since the con-
19 struction of this instance have since destructed.

20 The persona supplied to this instance's constructor is popped from this thread's
21 active stack. If this instance was constructed with the `mutex` constructor, then
22 that `mutex` is unlocked.

23 *UPC++ progress level:* none

24 `persona_scope& top_persona_scope();`

25 Reference to the most recently constructed but not destructed `persona_scope`
26 for this thread. Every thread begins with an implicitly instantiated scope point-
27 ing to its default persona that survives for the duration of the thread's lifetime.

28 *UPC++ progress level:* none

```
1 persona_scope& default_persona_scope();
```

2 Every thread begins with an implicitly instantiated scope pointing to its default
3 persona that survives for the duration of the thread's lifetime. This function
4 returns a reference to that bottommost `persona_scope` for the calling thread,
5 which points at the calling thread's `default_persona()`.

6 *UPC++ progress level: none*

7 10.5.3 Outgoing Progress

```
8 bool progress_required(persona_scope &ps = top_persona_scope());
```

9 *Precondition:* `ps` has been constructed by this thread.

10 For the set of personas included in this thread's active stack section bounded
11 inclusively between `ps` and the current top, *nearly* answers if any UPC++ op-
12 erations initiated by those personas require further advancement of internal-
13 progress of their respective personas before their completion events will be
14 eventually available to user-level progress on the destined ranks. The exact
15 meaning of the return value depends on which personas are selected by `ps`:

- 16 • If `ps` *does not* include the master persona: A return value of `true` means
17 that one or more of the personas indicated by `ps` requires further internal-
18 progress to achieve completion of its outgoing operations. A value of `false`
19 means that none of the personas indicated by `ps` require internal-progress,
20 but internal-progress of the master persona might still be required.
- 21 • If `ps` *does* include the master persona: A return value of `true` means that
22 one or more of the personas indicated by `ps` requires further internal-
23 progress to achieve completion of its outgoing operations. A return value
24 of `false` means that none of the non-master personas indicated by `ps`
25 requires further internal-progress, but the master persona may or may not
26 require further internal-progress.

27 *UPC++ progress level: none*

```
28 void discharge(persona_scope &ps = top_persona_scope());
```

29 Advances internal-progress enough to ensure that `progress_required(ps)` re-
30 turns `false`.

31 *UPC++ progress level: internal*

1 Chapter 11

2 Atomics

3 11.1 Overview

4 UPC++ supports atomic operations on shared memory locations. Atomicity entails that a
5 read-modify-write sequence on a memory location will happen without interference or inter-
6 leaving with other concurrently executing atomic operations. Atomicity is not guaranteed
7 if a memory location is concurrently targeted by both atomic and non-atomic operations.
8 The order in which concurrent atomics update the same memory is not guaranteed, not
9 even for successively issued operations by a single rank. Ordering of atomics with respect
10 to other asynchronous operations is also not guaranteed. The only means to ensure such
11 ordering is by waiting for one operation to complete before initiating its successor. Note
12 that UPC++ atomics do not interoperate with `std::atomic`.

13 At this time, it is unclear how UPC++ will support mixing of atomic and non-atomic
14 accesses to the same memory location. Until this is resolved, users must assume that
15 for the duration of the program, once a memory location is accessed via a UPC++ atomic,
16 only further atomic operations to that location will have meaningful results (note that even
17 global barrier synchronization does not grant an exception to this rule). This unfortunately
18 implies that deallocation of such memory is unsafe, as that would allow the memory to be
19 reallocated to a context unaware of its constrained condition.

20 All atomic operations are associated with an *atomic domain*. An atomic domain is
21 defined for an integer type and a set of operations. Currently, the allowed types are
22 `std::int32_t`, `std::uint32_t`, `std::int64_t`, and `std::uint64_t`. The list of oper-
23 ations is detailed in the API section below. The atomic domain is an instance of a
24 `atomic_domain` class, and the operations are defined as methods on that class.

25 The use of atomic domains permits selection (at construction) of the most efficient
26 available implementation which can provide correct results for the given set of operations
27 on the given data type. This is important because the best possible implementation of a

1 operation "X" may not be compatible with operation "Y". So, this best "X" can only be
2 used when it is known that "Y" will not be used. This issue arises because a NIC may
3 offload "X" (but not "Y") and use of a CPU-based implementation of "Y" would not be
4 coherent with the NIC performing a concurrent "X" operation.

5 Similar to a mutex, an atomic domain exists independent of the data it applies to.
6 User code is responsible for ensuring that data accessed via a given atomic domain is only
7 accessed via that domain, never via a different domain or without use of a domain.

8 Users may create as many domains as needed to describe their uses of atomic operations,
9 so long as there is at most one domain per atomic datum. If distinct data of the same type
10 are accessed using differing sets of operations, then creation of distinct domains for each
11 operation set is recommended to achieve the best performance on each set.

12 For example, to use atomic fetch-and-add, load and store operations on an `int64_t`, a
13 user must first define a domain as follows:

```
14 atomic_domain<int64_t> ad_i64({atomic_op::load,
15                               atomic_op::store,
16                               atomic_op::fetch_add});
```

17 Each atomic operation works on a *global pointer* to the integer type given when the
18 domain was constructed.

19 All atomic operations are non-blocking and provide an operation-completion event to
20 indicate completion of the atomic. By default, all operations return futures. So, for
21 example, this is the way to call an atomic operation for the previous example's domain:

```
22 global_ptr<int64_t> x = new_<int64_t>(0);
23 future<int64_t> f = ad_i64.fetch_add(x, 2,
24                                   std::memory_order_relaxed);
25 int64_t res = f.wait();
```

26 Atomic domains enable a user to select a subset of operations that are supported in
27 hardware on a given platform, and hence more performant.

28 11.2 API Reference

```
29 enum class atomic_op : int {
30     load, store,
31     add, fetch_add,
32     sub, fetch_sub,
33     inc, fetch_inc,
34     dec, fetch_dec,
35     compare_exchange
36 };
```

```
1  template<typename T>
2  class atomic_domain;
3      C++ Concepts: DefaultConstructible, MoveConstructible, MoveAssignable,
4      Destructible

5  template<typename T>
6  atomic_domain<T>::atomic_domain();

7      Precondition: T must be one of the approved atomic types: std::int32_t,
8      std::uint32_t, std::int64_t, or std::uint64_t.
9      Constructs an empty domain. This domain is not yet functional; it must be
10     overwritten through a move assignment.
11     This function may be called when UPC++ is in the uninitialized state.
12     UPC++ progress level: none
```

```
13 template<typename T>
14 atomic_domain<T>::atomic_domain(
15     std::vector<atomic_op> const &ops,
16     team &team = world());

17     Precondition: T must be one of the approved atomic types: std::int32_t,
18     std::uint32_t, std::int64_t, or std::uint64_t.
19     Constructs an atomic domain for integer type T, with supported operations
20     ops.
21     This constructor is a collective function over the specified team.
22     UPC++ progress level: internal
```

```
23 template<typename T>
24 atomic_domain<T>::~~atomic_domain();

25     Precondition: All operations initiated on the atomic domain must be complete
26     prior to any rank making this call (or the behavior is undefined). In prac-
27     tice, this means completing (syncing) all atomic operation at their initiators,
28     followed by a barrier prior to calling this function.
29     Destructs this atomic domain object.
30     This destructor is a collective function over the team used in the constructor.
31     UPC++ progress level: internal
```

```

1  template<typename T,
2      typename Completions=decltype(operation_cx::as_future())>
3  RType atomic_domain<T>::load(global_ptr<T> p,
4                              std::memory_order order,
5                              Completions cxs=Completions{});

```

6 *Precondition:* p must reference a valid object of type T. T must be the only type
7 used by any atomic referencing any part of p's target memory for the entire life-
8 time of UPC++. order must be std::memory_order_relaxed or std::memory_
9 order_acquire. The atomic_op::load operation must have been included in
10 the ops used to construct this atomic_domain.

11 Initiates an atomic read of the object at location p and produces its value as
12 part of operation completion.

13 *Completions:*

- 14 • *Operation:* Indicates completion of all aspects of the operation: the remote
15 atomic read and transfer of the result are complete. This completion
16 produces a value of type T.

17 *C++ memory ordering:* If order is std::memory_order_acquire then the
18 read performed will have a *happens-before* relationship with the operation-
19 completion notification actions (future readying, promise fulfillment, or persona
20 LPC enlistment).

21 *UPC++ progress level:* internal

```

22 template<typename T,
23     typename Completions=decltype(operation_cx::as_future())>
24 RType atomic_domain<T>::store(global_ptr<T> p,
25                               T val,
26                               std::memory_order order,
27                               Completions cxs=Completions{});

```

28 *Precondition:* p must reference a valid object of type T. T must be the only
29 type used by any atomic referencing any part of p's target memory for the
30 entire lifetime of UPC++. order must be std::memory_order_relaxed or std::
31 memory_order_release. The atomic_op::store operation must have been
32 included in the ops used to construct this atomic_domain.

1 Initiates an atomic write of `val` to the location `p`. Completion of the write is
2 indicated by operation completion.

3 *Completions:*

- 4 • *Operation:* Indicates completion of all aspects of the operation: the trans-
5 fer of the value and remote atomic write are complete.

6 *C++ memory ordering:* If `order` is `std::memory_order_release` then all eval-
7 uations *sequenced-before* this call will have a *happens-before* relationship with
8 the write performed. The write performed will have a *happens-before* rela-
9 tionship with the operation-completion notification actions (future readying,
10 promise fulfillment, or persona LPC enlistment).

11 *UPC++ progress level:* `internal`

```
12 template<typename T,  
13     typename Completions=decltype(operation_cx::as_future())>  
14 Type atomic_domain<T>::add(global_ptr<T> p,  
15     T val,  
16     std::memory_order order,  
17     Completions cxs=Completions{});
```

18 *Precondition:* `p` must reference a valid object of type `T`. `T` must be the only type
19 used by any atomic referencing any part of `p`'s target memory for the entire life-
20 time of UPC++. `order` must be `std::memory_order_relaxed`, `std::memory_`
21 `order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_`
22 `rel`. The `atomic_op::add` operation must have been included in the `ops` used
23 to construct this `atomic_domain`.

24 Initiates the atomic read-modify-write operation consisting of: reading the
25 value of the object located at `p`, adding `val` to it, and writing the new value
26 back.

27 *Completions:*

- 28 • *Operation:* Indicates completion of all aspects of the operation: the trans-
29 fer of the given value to the recipient, and remote atomic update are
30 complete.

31 *C++ memory ordering:* If `order` is either `std::memory_order_release` or
32 `std::memory_order_acq_rel` then all evaluations *sequenced-before* this call
33 will have a *happens-before* relationship with the atomic action. If `order` is `std::`

1 `memory_order_acquire` or `std::memory_order_acq_rel` then the atomic ac-
 2 tion will have a *happens-before* relationship with the operation-completion no-
 3 tification actions (future readying, promise fulfillment, or persona LPC enlist-
 4 ment).

5 *UPC++ progress level: internal*

```
6 template<typename T,
7     typename Completions=decltype(operation_cx::as_future())>
8 Type atomic_domain<T>::fetch_add(global_ptr<T> p,
9     T val,
10    std::memory_order order,
11    Completions cxs=Completions{});
```

12 *Precondition:* `p` must reference a valid object of type `T`. `T` must be the only type
 13 used by any atomic referencing any part of `p`'s target memory for the entire life-
 14 time of UPC++. `order` must be `std::memory_order_relaxed`, `std::memory_`
 15 `order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_`
 16 `rel`. The `atomic_op::fetch_add` operation must have been included in the
 17 `ops` used to construct this `atomic_domain`.

18 Initiates the atomic read-modify-write operation consisting of: reading the
 19 value of the object located at `p`, adding `val` to it, and writing the new value
 20 back. The value produced by operation completion is the one initially read.

21 *Completions:*

- 22 • *Operation:* Indicates completion of all aspects of the operation: the trans-
 23 fer of the given value to the recipient, remote atomic update, and transfer
 24 of the old value to the initiator are complete. This completion produces a
 25 value of type `T`.

26 *C++ memory ordering:* If `order` is either `std::memory_order_release` or
 27 `std::memory_order_acq_rel` then all evaluations *sequenced-before* this call
 28 will have a *happens-before* relationship with the atomic action. If `order` is `std::`
 29 `memory_order_acquire` or `std::memory_order_acq_rel` then the atomic ac-
 30 tion will have a *happens-before* relationship with the operation-completion no-
 31 tification actions (future readying, promise fulfillment, or persona LPC enlist-
 32 ment).

33 *UPC++ progress level: internal*


```
1  template<typename T,  
2      typename Completions=decltype(operation_cx::as_future()))>  
3  Type atomic_domain<T>::sub(global_ptr<T> p,  
4      T val,  
5      std::memory_order order,  
6      Completions cxs=Completions{});
```

7 *Precondition:* `p` must reference a valid object of type `T`. `T` must be the only type
8 used by any atomic referencing any part of `p`'s target memory for the entire life-
9 time of UPC++. `order` must be `std::memory_order_relaxed`, `std::memory_`
10 `order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_`
11 `rel`. The `atomic_op::sub` operation must have been included in the `ops` used
12 to construct this `atomic_domain`.

13 Initiates the atomic read-modify-write operation consisting of: reading the
14 value of the object located at `p`, subtracting `val` from it, and writing the new
15 value back.

16 *Completions:*

- 17 • *Operation:* Indicates completion of all aspects of the operation: the trans-
18 fer of the given value to the recipient, and remote atomic update are
19 complete.

20 *C++ memory ordering:* If `order` is either `std::memory_order_release` or
21 `std::memory_order_acq_rel` then all evaluations *sequenced-before* this call
22 will have a *happens-before* relationship with the atomic action. If `order` is `std::`
23 `memory_order_acquire` or `std::memory_order_acq_rel` then the atomic ac-
24 tion will have a *happens-before* relationship with the operation-completion no-
25 tification actions (future readying, promise fulfillment, or persona LPC enlist-
26 ment).

27 *UPC++ progress level:* `internal`

```
28  template<typename T,  
29      typename Completions=decltype(operation_cx::as_future()))>  
30  Type atomic_domain<T>::fetch_sub(global_ptr<T> p,  
31      T val,  
32      std::memory_order order,  
33      Completions cxs=Completions{});
```

1 *Precondition:* `p` must reference a valid object of type `T`. `T` must be the only type
 2 used by any atomic referencing any part of `p`'s target memory for the entire life-
 3 time of UPC++. `order` must be `std::memory_order_relaxed`, `std::memory_`
 4 `order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_`
 5 `rel`. The `atomic_op::fetch_sub` operation must have been included in the
 6 `ops` used to construct this `atomic_domain`.

7 Initiates the atomic read-modify-write operation consisting of: reading the
 8 value of the object located at `p`, subtracting `val` from it, and writing the new
 9 value back. The value produced by operation completion is the one initially
 10 read.

11 *Completions:*

- 12 • *Operation:* Indicates completion of all aspects of the operation: the trans-
 13 fer of the given value to the recipient, remote atomic update, and transfer
 14 of the old value to the initiator are complete. This completion produces a
 15 value of type `T`.

16 *C++ memory ordering:* If `order` is either `std::memory_order_release` or
 17 `std::memory_order_acq_rel` then all evaluations *sequenced-before* this call
 18 will have a *happens-before* relationship with the atomic action. If `order` is `std::`
 19 `memory_order_acquire` or `std::memory_order_acq_rel` then the atomic ac-
 20 tion will have a *happens-before* relationship with the operation-completion no-
 21 tification actions (future readying, promise fulfillment, or persona LPC enlist-
 22 ment).

23 *UPC++ progress level:* `internal`

```
24 template<typename T,  
25     typename Completions=decltype(operation_cx::as_future())>  
26 Type atomic_domain<T>::inc(global_ptr<T> p,  
27     std::memory_order order,  
28     Completions cxs=Completions{});
```

29 *Precondition:* `p` must reference a valid object of type `T`. `T` must be the only type
 30 used by any atomic referencing any part of `p`'s target memory for the entire life-
 31 time of UPC++. `order` must be `std::memory_order_relaxed`, `std::memory_`
 32 `order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_`
 33 `rel`. The `atomic_op::inc` operation must have been included in the `ops` used
 34 to construct this `atomic_domain`.

1 Initiates the atomic read-modify-write operation consisting of: reading the
2 value of the object located at `p`, adding 1 to it, and writing the new value
3 back.

4 *Completions:*

- 5 • *Operation:* Indicates completion of all aspects of the operation: the trans-
6 fer of the given value to the recipient, and remote atomic update are
7 complete.

8 *C++ memory ordering:* If `order` is either `std::memory_order_release` or
9 `std::memory_order_acq_rel` then all evaluations *sequenced-before* this call
10 will have a *happens-before* relationship with the atomic action. If `order` is `std::`
11 `memory_order_acquire` or `std::memory_order_acq_rel` then the atomic ac-
12 tion will have a *happens-before* relationship with the operation-completion no-
13 tification actions (future readying, promise fulfillment, or persona LPC enlist-
14 ment).

15 *UPC++ progress level:* `internal`

```
16 template<typename T,  
17     typename Completions=decltype(operation_cx::as_future())>  
18 Type atomic_domain<T>::fetch_inc(global_ptr<T> p,  
19     std::memory_order order,  
20     Completions cxs=Completions{});
```

21 *Precondition:* `p` must reference a valid object of type `T`. `T` must be the only type
22 used by any atomic referencing any part of `p`'s target memory for the entire life-
23 time of UPC++. `order` must be `std::memory_order_relaxed`, `std::memory_`
24 `order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_`
25 `rel`. The `atomic_op::fetch_inc` operation must have been included in the
26 `ops` used to construct this `atomic_domain`.

27 Initiates the atomic read-modify-write operation consisting of: reading the
28 value of the object located at `p`, adding 1 to it, and writing the new value
29 back. The value produced by operation completion is the one initially read.

30 *Completions:*

- 31 • *Operation:* Indicates completion of all aspects of the operation: the trans-
32 fer of the given value to the recipient, remote atomic update, and transfer
33 of the old value to the initiator are complete. This completion produces a
34 value of type `T`.

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).

UPC++ progress level: `internal`

```

9  template<typename T,
10         typename Completions=decltype(operation_cx::as_future())>
11  Type atomic_domain<T>::dec(global_ptr<T> p,
12                             std::memory_order order,
13                             Completions cxs=Completions{});

```

Precondition: `p` must reference a valid object of type `T`. `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 `atomic_op::dec` operation must have been included in the `ops` used to construct this `atomic_domain`.

Initiates the atomic read-modify-write operation consisting of: reading the value of the object located at `p`, subtracting 1 from it, and writing the new value back.

Completions:

- *Operation:* Indicates completion of all aspects of the operation: the transfer of the given value to the recipient, and remote atomic update are complete.

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).

UPC++ progress level: `internal`

```
1  template<typename T,  
2      typename Completions=decltype(operation_cx::as_future())>  
3  Type atomic_domain<T>::fetch_dec(global_ptr<T> p,  
4      std::memory_order order,  
5      Completions cxs=Completions{});
```

6 *Precondition:* `p` must reference a valid object of type `T`. `T` must be the only type
7 used by any atomic referencing any part of `p`'s target memory for the entire life-
8 time of UPC++. `order` must be `std::memory_order_relaxed`, `std::memory_`
9 `order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_`
10 `rel`. The `atomic_op::fetch_dec` operation must have been included in the
11 `ops` used to construct this `atomic_domain`.

12 Initiates the atomic read-modify-write operation consisting of: reading the
13 value of the object located at `p`, subtracting 1 from it, and writing the new
14 value back. The value produced by operation completion is the one initially
15 read.

16 *Completions:*

- 17 • *Operation:* Indicates completion of all aspects of the operation: the trans-
18 fer of the given value to the recipient, remote atomic update, and transfer
19 of the old value to the initiator are complete. This completion produces a
20 value of type `T`.

21 *C++ memory ordering:* If `order` is either `std::memory_order_release` or
22 `std::memory_order_acq_rel` then all evaluations *sequenced-before* this call
23 will have a *happens-before* relationship with the atomic action. If `order` is `std::`
24 `memory_order_acquire` or `std::memory_order_acq_rel` then the atomic ac-
25 tion will have a *happens-before* relationship with the operation-completion no-
26 tification actions (future readying, promise fulfillment, or persona LPC enlist-
27 ment).

28 *UPC++ progress level:* **internal**

```
29  template<typename T,  
30      typename Completions=decltype(operation_cx::as_future())>  
31  Type atomic_domain<T>::compare_exchange(  
32      global_ptr<T> p,  
33      T val1,  
34      T val2,  
35      std::memory_order order,  
36      Completions cxs=Completions{});
```

1 *Precondition:* `p` must reference a valid object of type `T`. `T` must be the only type
2 used by any atomic referencing any part of `p`'s target memory for the entire life-
3 time of UPC++. `order` must be `std::memory_order_relaxed`, `std::memory_`
4 `order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_`
5 `rel`. The `atomic_op::compare_exchange` operation must have been included
6 in the `ops` used to construct this `atomic_domain`.

7 Initiates the atomic read-modify-write operation consisting of: reading the
8 value of the object located at `p`, and if it is equal to `val1`, writing `val2` back.
9 The value produced by operation completion is the one initially read.

10 *Completions:*

- 11 • *Operation:* Indicates completion of all aspects of the operation: the trans-
12 fer of the given value to the recipient, remote atomic update, and transfer
13 of the old value to the initiator are complete. This completion produces a
14 value of type `T`.

15 *C++ memory ordering:* If `order` is either `std::memory_order_release` or
16 `std::memory_order_acq_rel` then all evaluations *sequenced-before* this call
17 will have a *happens-before* relationship with the atomic action. If `order` is `std::`
18 `memory_order_acquire` or `std::memory_order_acq_rel` then the atomic ac-
19 tion will have a *happens-before* relationship with the operation-completion no-
20 tification actions (future readying, promise fulfillment, or persona LPC enlist-
21 ment).

22 *UPC++ progress level:* `internal`

1 Chapter 12

2 Teams

3 12.1 Overview

4 UPC++ provides *teams* as a means of grouping ranks. UPC++ uses `teams` for collective op-
5 erations. `team` construction is collective and should be considered moderately expensive
6 and done as part of the set-up phase of a calculation. `teams` are similar to `MPI_Groups`
7 and the default `team` is `world()`. `teams` are considered special when it comes to serial-
8 ization. Each `team` has a unique `team_id` that is equal across the `team` and acts as an
9 opaque handle. Any rank that is a member of the `team` can retrieve the `team` object with
10 the `team_id::here()` function. Hence, coordinating ranks can reference specific `teams` by
11 their `team_id`.

12 While a rank within a UPC++ SPMD program can have multiple `inrank_t` values that
13 represent their relative placement in several `teams`, it is the `inrank_t` in the `world()` that
14 is used in all UPC++ functions, unless otherwise specifically noted. For example, `broadcast`
15 uses the team-relative rank.

16 12.2 Local Teams

17 Each rank can obtain a reference to a special team by calling `local_team.global_ptr`'s to
18 objects allocated by ranks within this `team` will report `is_local() == true` and `local()`
19 will return a valid `T*` to that memory. The `global_ptr.where()` function will report the
20 rank (in team `world()`) that originally acquired that memory using the functions in chapter
21 4. It is not guaranteed that the `T*`'s obtained by different ranks to the same shared object
22 will have bit-wise identical pointer values. In the general case, peers may have different
23 virtual addresses for the same physical memory.

1 12.3 API Reference

2 12.3.1 team

```
3 class team;
```

4 C++ Concepts: MoveConstructible, Destructible

```
5 intrank_t team::rank_n() const;
```

6 Returns the number of ranks that are in the given team.

7 *UPC++ progress level: none*

```
8 intrank_t team::rank_me() const;
```

9 Returns the peer index of the caller in the given team.

10 *UPC++ progress level: none*

```
11 intrank_t team::operator[](intrank_t peer_index) const;
```

12 *Precondition: peer_index >= 0 and peer_index < rank_n().*

13 Returns the index in the world() team for the rank associated with peer_index
14 in this team.

15 *UPC++ progress level: unspecified between none and internal*

```
16 intrank_t team::from_world(inrank_t world_index) const;
```

```
17 intrank_t team::from_world(inrank_t world_index,  
18 intrank_t otherwise) const;
```

19 *Precondition: world_index >= 0 and world_index < world().rank_n().* For
20 the single argument overload, the rank associated with world_index must be
21 a member of this team.

22 Returns the peer index in this team of the rank associated with world_index in
23 the world() team. For the two argument overload, if the rank is not a member
24 of this team then the value of otherwise is returned.

25 *UPC++ progress level: unspecified between none and internal*


```
1 team team::split(intrank_t color, intrank_t key);
```

2 *Precondition:* This function must be called collectively by all the ranks in this
3 team, and it must be called by the thread that has the master persona (§10.5.1).
4 No two ranks in the collective call may specify the same combination of `color`
5 and `key`.

6 Splits the given team into subteams based on the `color` and `key` arguments.
7 All ranks that call the function with the same `color` value will be separated
8 into the same subteam. Ranks in the same subteam will be numbered according
9 to their position in the sequence of sorted key values. The return value is the
10 team representing the calling rank's new subteam. This call will invoke user-
11 level progress, so the caller may expect incoming RPCs to fire before it returns.

12 *C++ memory ordering:* With respect to all threads participating in this col-
13 lective, all evaluations which are *sequenced-before* their respective thread's in-
14 vocation of this call will have a *happens-before* relationship with all evaluations
15 sequenced after the call.

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

```
17 team::team(team &&other);
```

18 *Precondition:* Calling thread must have the master persona.

19 Makes this instance the calling rank's representative of the team associated with
20 `other`, transferring all state from `other`. Invalidates `other`, and any subsequent
21 operations on `other`, except for destruction, produce undefined behavior.

22 *UPC++ progress level:* `none`

```
23 team::~~team();
```

24 *Precondition:* Calling thread must have the master persona.

25 If this instance has not been invalidated by being passed to the move construc-
26 tor, then this will destroy the current rank's state associated with the team.
27 Further lookups on this rank using the `team_id` corresponding to this team will
28 have undefined behavior. If this instance has been invalidated by a move, then
29 this call will have no effect.

30 *UPC++ progress level:* `none`

```
31 team_id team::id() const;
```

32 Returns the universal name associated with this team.

33 *UPC++ progress level:* `none`

1 12.3.2 team_id

2 `class team_id;`

3 C++ Concepts: DefaultConstructible, TriviallyCopyable, StandardLayoutType,
4 EqualityComparable, LessThanComparable, hashable

5 UPC++ Concepts: DefinitelyTriviallySerializable

6 A universal name representing a team.

7 `team_id::team_id();`

8 Initializes this name to be an invalid ID.

9 *UPC++ progress level: none*

10 `team& team_id::here() const;`

11 *Precondition:* This name must be a valid ID. The current rank must be a
12 member of the `team` associated with this name, and it must have completed
13 creation of the `team`. The `team` must not have been destroyed.

14 Retrieves a reference to the `team` instance associated with this name.

15 *UPC++ progress level: none*

16 `future<team &> team_id::when_here() const;`

17 *Precondition:* This name must be a valid ID. The current rank must be a
18 member of the `team` associated with this name. The calling thread must have
19 the master persona. The `team` must not have been destroyed.

20 Retrieves a future representing when the current rank constructs the `team` cor-
21 responding to this name.

22 *UPC++ progress level: none*

1 12.3.3 Fundamental Teams

2 `team& world();`

3 Returns a reference to the team representing all the ranks in the program. The
4 result is undefined if a move is performed on the returned team.

5 *UPC++ progress level: none*

6 `inrank_t rank_n();`

7 Returns the number of ranks that are in the world team.

8 Equivalent to: `world().rank_n()`.

9 *UPC++ progress level: none*

10 `inrank_t rank_me();`

11 Returns the peer index of the caller in the world team.

12 Equivalent to: `world().rank_me()`.

13 *UPC++ progress level: none*

14 `team& local_team();`

15 Returns a reference to the local team containing this rank. A local team rep-
16 represents a set of ranks which share physical memory (§12.2). The result is
17 undefined if a move is performed on the returned team.

18 *UPC++ progress level: none*

19 `bool local_team_contains(inrank_t world_index);`

20 *Precondition:* `world_index >= 0` and `world_index < world().rank_n()`.

21 Determines if `world_index` is a member of the local team containing the this
22 rank (§12.2).

23 Equivalent to: `local_team().from_world(world_index,-1) >= 0`

24 *UPC++ progress level: none*

1 12.3.4 Collectives

```
2 void barrier(team &team = world());
```

3 *Precondition:* This function must be called collectively by all the ranks in the
4 given team, and it must be called by the thread that has the master persona
5 (§10.5.1).

6 Performs a barrier operation over the given team. The call will not return until
7 all ranks in the team have entered the call. There is no implied relationship
8 between this call and other in-flight operations. This call will invoke user-level
9 progress, so the caller may expect incoming RPCs to fire before it returns.

10 *C++ memory ordering:* With respect to all threads participating in this col-
11 lective, all evaluations which are *sequenced-before* their respective thread's in-
12 vocation of this call will have a *happens-before* relationship with all evaluations
13 sequenced after the call.

14 *UPC++ progress level:* **user**

```
15 template<typename Completions=decltype(operation_cx::as_future())>  
16 RType barrier_async(team &team = world(),  
17                    Completions cxs=Completions{});
```

18 *Precondition:* This function must be called collectively by all the ranks in the
19 given team, and it must be called by the thread that has the master persona
20 (§10.5.1).

21 Initiates an asynchronous barrier operation over the given team. The call will
22 return without waiting for other ranks to make the call. Operation completion
23 will be signaled after all other ranks in the team have entered the call.

24 *Completions:*

- 25 • *Operation:* Indicates completion of the collective from the viewpoint of
26 the caller, implying that all ranks in the given team have entered the
27 collective.

28 *C++ memory ordering:* With respect to all threads participating in this col-
29 lective, all evaluations which are *sequenced-before* their respective thread's in-
30 vocation of this call will have a *happens-before* relationship with all evaluations
31 sequenced after the operation-completion notification actions (future readying,
32 promise fulfillment, or persona LPC enlistment).

33 *UPC++ progress level:* **internal**

```
1  template<typename T, typename BinaryOp,
2          typename Completions=decltype(operation_cx::as_future())>
3  RType allreduce(T &&value, BinaryOp &&op, team &team = world(),
4                 Completions cxs=Completions{});
```

5 *Precondition:* This function must be called collectively by all the ranks in the
6 given team, and it must be called by the thread that has the master per-
7 sona (§10.5.1). T must be DefinitelyTriviallySerializable. BinaryOp must be a
8 function-object type representing an associative and commutative mathemat-
9 ical operation taking two values of type T and returning a value implicitly
10 convertible to T. BinaryOp must be referentially transparent and concurrently
11 invocable. BinaryOp may not invoke any UPC++ routine with a progress level
12 other than none.

13 Performs a reduction operation over the ranks in the given team. If the team
14 contains only a single rank, then the resulting operation completion will produce
15 value. Otherwise, initiates an asynchronous reduction over the values provided
16 by each rank. The reduction is performed in some non-deterministic order by
17 applying op to combine values and intermediate results. Each rank receives the
18 result of the reduction as part of operation completion.

19 *Completions:*

- 20 • *Operation:* Indicates completion of the collective from the viewpoint of
21 the caller, implying that the result of the reduction is available at this
22 rank. This completion produces a value of type T.

23 *C++ memory ordering:* With respect to all threads participating in this col-
24 lective, all evaluations which are *sequenced-before* their respective thread's in-
25 vocation of this call will have a *happens-before* relationship with all evaluations
26 sequenced after the operation-completion notification actions (future readying,
27 promise fulfillment, or persona LPC enlistment).

28 *UPC++ progress level:* **internal**

```
29  template<typename T,
30          typename Completions=decltype(operation_cx::as_future())>
31  RType broadcast(T &&value, intrank_t sender,
32                 team &team = world(),
33                 Completions cxs=Completions{});
34
35  template<typename T,
36          typename Completions=decltype(operation_cx::as_future())>
```

```
1 RType broadcast(T *buffer, std::size_t count,  
2               intrank_t sender, team &team = world(),  
3               Completions cxs=Completions{});
```

4 *Precondition:* The function must be called collectively by the ranks in the
5 given team, and it must be called by the thread that has the master persona
6 (§10.5.1). The value of `sender`, and `count` in the second variant, must be the
7 same across all callers. In the second variant, the addresses in the interval
8 `[buffer,buffer+count)` must all reference valid objects of type `T`. The type
9 `T` must be `DefinitelyTriviallySerializable`.

10 Initiates an asynchronous broadcast (one-to-all) operation, with rank `sender`
11 of `team` acting as the producer of the broadcast. In the first variant, `value`
12 will be asynchronously sent to all ranks in the team, encapsulated in operation
13 completion, which will be signaled upon receipt of the value. In the second
14 variant, the objects in `[buffer,buffer+count)` on rank `sender` are sent to the
15 addresses `[buffer,buffer+count)` provided by the receiving ranks. Operation
16 completion signals completion of the operation with respect to the calling rank.
17 For the sender, this indicates that the given buffer is available for reuse, and
18 for a receiver, it indicates that the data have been received in its buffer.

19 *Completions:*

- 20 • *Operation:* In the first variant, indicates that the value provided by the
21 sender is available at the caller. This completion produces a value of type
22 `T`.

23 In the second variant, indicates completion of the collective from the view-
24 point of the caller as described above.

25 *C++ memory ordering:* With respect to all threads participating in this col-
26 lective, all evaluations which are *sequenced-before* the producing thread's invo-
27 cation of this call will have a *happens-before* relationship with all evaluations
28 sequenced after the operation-completion notification actions (future readying,
29 promise fulfillment, or persona LPC enlistment).

30 *UPC++ progress level:* `internal`

1 Chapter 13

2 Distributed Objects

3 13.1 Overview

4 In distributed-memory parallel programming, the concept of a single logical object parti-
5 tioned over several ranks is a useful capability in many contexts: for example, geometric
6 meshes, vectors, matrices, tensors, and associative maps. Since `UPC++` is a communication
7 library, it strives to focus on the mechanisms of communication as opposed to the various
8 programming idioms for managing distribution. However, a basic framework for users to
9 implement their own distributed objects is useful and also enables `UPC++` to provide the
10 user with the following valuable features:

- 11 1. Universal distributed object naming: per-object names that can be transmitted to
12 other ranks while retaining their meaning.
- 13 2. Name-to-this mapping: Mapping between the universal name and the current rank's
14 memory address holding that distributed object's state for the rank (the current
15 rank's `this` pointer).

16 The need for universal distributed object naming stems primarily from `RPC`-based com-
17 munication. If one rank needs to remotely invoke code on a peer's partition of a distributed
18 object, there needs to be some mutually agreeable identifier for referring to that distributed
19 object. For simplicity, this identifier value should be: identical across all ranks so that it
20 may be freely communicated while maintaining its meaning. Moreover, the name should
21 be `TriviallyCopyable` so that it may be serialized into `RPCs` efficiently (including with the
22 auto-capture `[=]` lambda syntax), hashable, and comparable so that it works well with
23 standard `C++` containers. `UPC++` provides distributed object names meeting these criteria
24 as well as the registry for mapping names to and from the current rank's partition of the
25 distributed object.

13.2 Building Distributed Objects

Distributed objects are built with the `upcxx::dist_object<T>` type. For all ranks in a given team, each rank constructs an instance of `dist_object<T>`, supplying a value of type `T` representing this rank's instance value. All ranks in the team must call this constructor collectively. Once construction completes, the distributed object has a universal name which can be used on any rank in the team to locate the resident instance. 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 rank. Thus, the programmer should ensure that no rank destructs a 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.

The names of `dist_object<T>`'s are encoded by the `dist_id<T>` type. This type is `TriviallyCopyable`, `EqualityComparable`, `LessThanComparable`, `hashable`, and `DefinitelyTriviallySerializable`. It has the members `.here()` and `.when_here()` for retrieving the resident `dist_object<T>` instance registered with the name.

13.3 Ensuring Distributed Existence

The `dist_object<T>` constructor requires it be called in a collective context, but it does not guarantee that, after the call, all other ranks in the team have exited or even reached the constructor. Thus users are required to guard against the possibility that when an RPC carrying an distributed object's name executes, the recipient rank 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 created distributed object, the relevant team completes a `barrier` to ensure global existence of the `dist_object<T>`.
2. Point to point: Before communicating a `dist_id<T>` with a given rank, the initiating rank uses some two-party protocol to ensure that the peer has constructed 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, must wait asynchronously via a continuation for the peer to construct the distributed object.

UPC++ enables the asynchronous point-to-point approach implicitly when `dist_object<T>&` arguments are given to any of the RPC family of functions (see Ch. 9).

1 13.4 API Reference

```
2 template<typename T>  
3 class dist_object;
```

4 C++ Concepts: MoveConstructible, Destructible

```
5 template<typename T>  
6 dist_object<T>::dist_object(T value, team &team = world());
```

7 *Precondition:* Calling thread must have the master persona.

8 Constructs this rank's member of the distributed object identified by the col-
9 lective calling context across `team`. The initial value for this rank is given
10 in `value`. The future returned from `dist_id<T>::when_here` for the corre-
11 sponding `dist_id<T>` will be readied during this constructor. This implies
12 that continuations waiting for that future will execute before the constructor
13 returns.

14 *UPC++ progress level:* none

```
15 template<typename T>  
16 template<typename ...Arg>  
17 dist_object<T>::dist_object(team &team, Arg &&...arg);
```

18 *Precondition:* Calling thread must have the master persona.

19 Constructs this rank's member of the distributed object identified by the col-
20 lective calling context across `team`. The initial value for this rank is constructed
21 with `T(std::forward<Arg>(arg)...)...`. The result is undefined if this call
22 throws an exception. The future returned from `dist_id<T>::when_here` for
23 the corresponding `dist_id<T>` will be readied during this constructor. This
24 implies that continuations waiting for that future will execute before the con-
25 structor returns.

26 *UPC++ progress level:* none

```
27 template<typename T>  
28 dist_object<T>::dist_object(dist_object<T> &&other);
```

1 *Precondition:* Calling thread must have the master persona.
 2 Makes this instance the calling rank’s representative of the distributed object
 3 associated with `other`, transferring all state from `other`. Invalidates `other`, and
 4 any subsequent operations on `other`, except for destruction, produce undefined
 5 behavior.
 6 *UPC++ progress level:* none

```
7  template<typename T>
8  dist_object<T>::~dist_object();
```

9 *Precondition:* Calling thread must have the master persona.
 10 If this instance has not been invalidated by being passed to the move construc-
 11 tor, then this will destroy the current rank’s member of the distributed object.
 12 `~T()` will be invoked on the resident instance, and further lookups on this rank
 13 using the `dist_id<T>` corresponding to this distributed object will have unde-
 14 fined behavior. If this instance has been invalidated by a move, then this call
 15 will have no effect.
 16 *UPC++ progress level:* none

```
17 template<typename T>
18 dist_id<T> dist_object<T>::id() const;
```

19 Returns the `dist_id<T>` representing the universal name of this distributed
 20 object.
 21 *UPC++ progress level:* none

```
22 template<typename T>
23 team& dist_object<T>::team() const;
```

24 *Precondition:* The `team` associated with this distributed object must not have
 25 been destroyed.
 26 Retrieves a reference to the `team` instance associated with this distributed ob-
 27 ject.
 28 *UPC++ progress level:* none

```
29 template<typename T>
30 T* dist_object<T>::operator->() const;
```

1 Access to the current rank's value instance for this distributed object.

2 *UPC++ progress level: none*

```
3 template<typename T>
4 T& dist_object<T>::operator*() const;
```

5 Access to the current rank's value instance for this distributed object.

6 *UPC++ progress level: none*

```
7 template<typename T>
8 future<T> dist_object<T>::fetch(intrank_t rank) const;
```

9 *Precondition:* `rank` must be a valid ID in the team associated with this dis-
10 tributed object. `T` must be `DefinitelySerializable` but not `view<U, IterType>`.
11 `rank`'s instance of this distributed object must not have been destroyed. The
12 `team` associated with this distributed object must not have been destroyed.

13 Asynchronously retrieves a copy of the instance of this distributed object asso-
14 ciated with the peer index `rank` in this distributed object's team. The result is
15 encapsulated in the returned future. This call is equivalent to

```
16     rpc(team()[rank],
17         [] (dist_object<T> &obj) { return *obj; },
18         *this)
```

19 *UPC++ progress level: internal*

```
20 template<typename T>
21 struct dist_id<T>;
```

22 C++ Concepts: `DefaultConstructible`, `TriviallyCopyable`, `StandardLayoutType`,
23 `EqualityComparable`, `LessThanComparable`, `hashable`

24 UPC++ Concepts: `DefinitelyTriviallySerializable`

```
25 template<typename T>
26 dist_id<T>::dist_id();
```

27 Initializes this name to be an invalid ID.

28 *UPC++ progress level: none*

```
1  template<typename T>
2  future<dist_object<T>&> dist_id<T>::when_here() const;
```

3 *Precondition:* This name must be a valid ID. The current rank's `dist_object<T>`
4 instance associated with this name must not have been destroyed. The calling
5 thread must have the master persona.

6 Retrieves a future representing when the current rank constructs the
7 `dist_object<T>` corresponding to this name.

8 *UPC++ progress level: none*

```
9  template<typename T>
10 dist_object<T>& dist_id<T>::here() const;
```

11 *Precondition:* This name must be a valid ID. The current rank's `dist_object<T>`
12 instance associated with this name must be alive. The calling thread must have
13 the master persona.

14 Retrieves a reference to the current rank's `dist_object<T>` instance associated
15 with this name.

16 *UPC++ progress level: none*

1 Chapter 14

2 Non-Contiguous One-Sided 3 Communication

4 14.1 Overview

5 UPC++ provides functions to perform one-sided communications similar to `rget` and `rput`
6 which are dedicated to handle data stored in non-contiguous locations. These functions are
7 denoted with a suffix added to the type of operation, in increasing order of specialization:

8 `{rput,rget}_{irregular,regular,strided}`

9 The most general variant of the API, `{rput,rget}_irregular`, accept iterators over
10 an array or collection of `std::pair` (or `std::tuple`) that contain a local or global pointer
11 to a memory location in the first member while the second member contains the size of
12 the contiguous chunk of memory to be transferred. This variant is capable of expressing
13 non-contiguous RMA of arbitrary shape, but pays the highest overhead in metadata to
14 payload ratio.

15 The next set of functions, `{rput,rget}_regular`, operates over contiguous elements of
16 identical size on each side of the transfer, and only requires the caller to provide an array
17 or collection of base pointers to each element.

18 Finally, the most specialized set of functions, `{rput,rget}_strided`, provide an in-
19 terface for expressing translational and transposing copies between arbitrary rectangular
20 sections of densely stored N-dimensional arrays. This specialized pattern requires the least
21 metadata, which is constant in size for a given dimensionality. An example of such a
22 transfer is depicted in Figure 14.1.

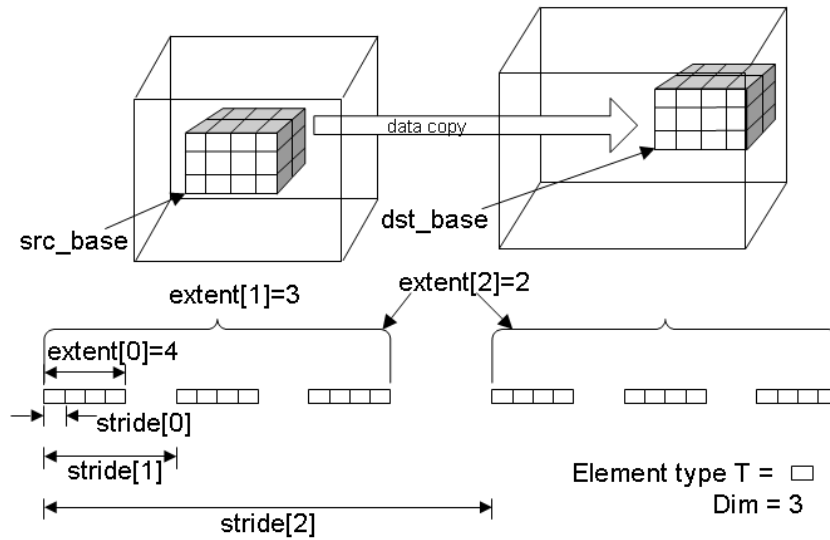


Figure 14.1: Example of a 3-D strided translational copy, with associated metadata

1 14.2 API Reference

2 14.2.1 Requirements on Iterators

3 An iterator used with a UPC++ operation in this section must adhere to the following
4 requirements:

- 5 • It must satisfy the Iterator and EqualityComparable C++ concepts.
- 6 • Calling `std::distance` on the iterator must not invalidate it.

7 14.2.2 Irregular Put

```
8 template<typename SrcIter, typename DestIter,
9         typename Completions=decltype(operation_cx::as_future())>
10 RType rput_irregular(
11     SrcIter src_runs_begin, SrcIter src_runs_end,
12     DestIter dest_runs_begin, DestIter dest_runs_end,
13     Completions cxs=Completions{});
```

14 *Preconditions:*

15 SrcIter and DestIter both satisfy the iterator requirements above.
16 `std::get<0>(*std::declval<SrcIter>())` has a return type convertible
17 to `T const*`, for some DefinitelyTriviallySerializable type T.

1 `std::get<1>(*std::declval<SrcIter>())` has a return type convertible
2 to `std::size_t`.

3 `std::get<0>(*std::declval<DestIter>())` has the return type
4 `global_ptr<T>`, for the same type `T` as with `SrcIter`.

5 `std::get<1>(*std::declval<DestIter>())` has a return type convert-
6 ible to `std::size_t`.

7 All destination addresses must be `global_ptr<T>`'s referencing memory
8 with affinity to the same rank.

9 The length of the expanded address sequence (the sum over the run
10 lengths) must be the same for the source and destination sequences.

11 For some type `T`, takes a sequence of source addresses of `T const*` and a se-
12 quence of destination addresses of `global_ptr<T>` and does the corresponding
13 puts from each source address to the destination address of the same sequence
14 position.

15 Address sequences are encoded in run-length form as sequences of runs, where
16 each run is a pair consisting of a starting address plus the number of consecutive
17 elements of type `T` beginning at that address.

18 As an example of valid types for individual runs, `SrcIter` could be an iterator
19 over elements of type `std::pair<T const*, std::size_t>`, and `DestIter` an
20 iterator over `std::pair<global_ptr<T>, std::size_t>`. Variations replac-
21 ing `std::pair` with `std::tuple` or `size_t` with other primitive integral types
22 are also valid.

23 The source sequence iterators must remain valid, and the underlying addresses
24 and source memory contents must not be modified until source completion
25 is signaled. Only after source completion is signaled can the source address
26 sequences and memory be reclaimed by the application.

27 The destination sequence iterators must remain valid until source completion
28 is signaled.

29 The destination memory regions must be completely disjoint and must not over-
30 lap with any source memory regions, otherwise behavior is undefined. Source
31 regions are permitted to overlap with each other.

32 *Completions:*

- 33 • *Source:* Indicates that the source sequence iterators and underlying mem-
34 ory, as well as the destination sequence iterators, are no longer in use by
35 UPC++ and may be reclaimed by the user.

- 1 • *Remote*: Indicates completion of the transfer of all values.
- 2 • *Operation*: Indicates completion of all aspects of the operation: the trans-
3 fer and remote stores are complete.

4 *C++ memory ordering*: The reads of the sources will have a *happens-before*
5 relationship with the source-completion notification actions (future readying,
6 promise fulfillment, or persona LPC enlistment). The writes to the destinations
7 will have a *happens-before* relationship with the operation-completion notifica-
8 tion actions (future readying, promise fulfillment, or persona LPC enlistment)
9 and remote-completion actions (RPC enlistment). For LPC and RPC com-
10 pletions, all evaluations *sequenced-before* this call will have a *happens-before*
11 relationship with the execution of the completion function.

12 *UPC++ progress level*: **internal**

13 14.2.3 Irregular Get

```
14 template<typename SrcIter, typename DestIter,  
15           typename Completions=decltype(operation_cx::as_future())>  
16 RType rget_irregular(  
17     SrcIter src_runs_begin, SrcIter src_runs_end,  
18     DestIter dest_runs_begin, DestIter dest_runs_end,  
19     Completions cxs=Completions{});
```

20 *Preconditions*:

21 SrcIter and DestIter both satisfy the iterator requirements above.

22 std::get<0>>(*std::declval<SrcIter>()) has the type global_ptr<T>
23 for some DefinitelyTriviallySerializable type T.

24 std::get<1>>(*std::declval<SrcIter>()) has a type that is convertible
25 to std::size_t.

26 std::get<0>>(*std::declval<DestIter>()) has the type T*, for the same
27 type T as with SrcIter.

28 std::get<1>>(*std::declval<DestIter>()) has a type that is convert-
29 ible to std::size_t.

30 All source addresses must be global_ptr<T>'s referencing memory with
31 affinity to the same rank.

32 The length of the expanded address sequence (the sum over the run
33 lengths) must be the same for the source and destination sequences.

1 For some type `T`, takes a sequence of source addresses of `global_ptr<T>` and a
2 sequence of destination addresses of `T*` and does the corresponding gets from
3 each source address to the destination address of the same sequence position.

4 Address sequences are encoded in run-length form as sequences of runs, where
5 each run is a pair consisting of a starting address plus the number of consecutive
6 elements of type `T` beginning at that address.

7 As an example of valid types for individual runs, `DestIter` could be an it-
8 erator over elements of type `std::pair<T*, std::size_t>`, and `SrcIter` an
9 iterator over `std::pair<global_ptr<T>, std::size_t>`. Variations replac-
10 ing `std::pair` with `std::tuple` or `size_t` with other primitive integral types
11 are also valid.

12 The source sequence iterators must remain valid, and the underlying addresses
13 and memory contents must not be modified until operation completion is sig-
14 naled. Only after operation completion is signaled can the address sequences
15 and source memory be reclaimed by the application.

16 The destination sequence iterators must remain valid until operation comple-
17 tion is signaled.

18 The destination memory regions must be completely disjoint and must not over-
19 lap with any source memory regions, otherwise behavior is undefined. Source
20 regions are permitted to overlap with each other.

21 *Completions:*

- 22 • *Operation:* Indicates completion of all aspects of the operation: the trans-
23 fer and local stores are complete.

24 *C++ memory ordering:* The reads of the sources and writes to the destina-
25 tions will have a *happens-before* relationship with the operation-completion no-
26 tification actions (future readying, promise fulfillment, or persona LPC enlist-
27 ment). For LPC completions, all evaluations *sequenced-before* this call will have
28 a *happens-before* relationship with the execution of the completion function.

29 *UPC++ progress level:* `internal`

30 14.2.4 Regular Put

```
31 template<typename SrcIter, typename DestIter,  
32           typename Completions=decltype(operation_cx::as_future())>  
33 RType rput_regular(  
34     SrcIter src_runs_begin, SrcIter src_runs_end,
```

```
1  std::size_t src_run_length ,
2  DestIter dest_runs_begin , DestIter dest_runs_end ,
3  std::size_t dest_run_length ,
4  Completions cxs=Completions{ });
```

5 *Preconditions:*

6 SrcIter and DestIter both satisfy the iterator requirements above.

7 *std::declval<SrcIter>() has a type convertible to T const*, for some
8 DefinitelyTriviallySerializable type T.

9 *std::declval<DestIter>() has the type global_ptr<T>, for the same
10 type T as with SrcIter.

11 All destination addresses must be global_ptr<T>'s referencing memory
12 with affinity to the same rank.

13 The length of the two sequences delimited by (src_runs_begin,
14 src_runs_end) and (dest_runs_begin, dest_runs_end) multiplied by
15 src_run_length and dest_run_length, respectively, must be the same.

16 This call has the same semantics as rput_irregular with the exception that,
17 for each sequence, all run lengths are the same and are factored out of the
18 sequences into two extra parameters src_run_length and dest_run_length,
19 which express the number of consecutive elements of type T in units of element
20 count. Thus the iterated elements are no longer pairs, but just pointers.

21 The source sequence iterators must remain valid, and the underlying addresses
22 and source memory contents must not be modified until source completion
23 is signaled. Only after source completion is signaled can the source address
24 sequences and memory be reclaimed by the application.

25 The destination sequence iterators must remain valid until source completion
26 is signaled.

27 *Completions:*

- 28 • *Source:* Indicates that the source sequence iterators and underlying mem-
29 ory, as well as the destination sequence iterators, are no longer in use by
30 UPC++ and may be reclaimed by the user.
- 31 • *Remote:* Indicates completion of the transfer of all values.
- 32 • *Operation:* Indicates completion of all aspects of the operation: the trans-
33 fer and remote stores are complete.

34 *C++ memory ordering:* The reads of the sources will have a *happens-before*
35 relationship with the source-completion notification actions (future readying,

1 promise fulfillment, or persona LPC enlistment). The writes to the destinations
2 will have a *happens-before* relationship with the operation-completion notifica-
3 tion actions (future readying, promise fulfillment, or persona LPC enlistment)
4 and remote-completion actions (RPC enlistment). For LPC and RPC com-
5 pletions, all evaluations *sequenced-before* this call will have a *happens-before*
6 relationship with the execution of the completion function.

7 *UPC++ progress level: internal*

8 14.2.5 Regular Get

```
9 template<typename SrcIter, typename DestIter,  
10           typename Completions=decltype(operation_cx::as_future())>  
11 RType rget_regular(  
12     SrcIter src_runs_begin, SrcIter src_runs_end,  
13     std::size_t src_run_length,  
14     DestIter dest_runs_begin, DestIter dest_runs_end,  
15     std::size_t dest_run_length,  
16     Completions cxs=Completions{});
```

17 *Preconditions:*

18 SrcIter and DestIter both satisfy the iterator requirements above.

19 *std::declval<DestIter>() has a type convertible to T*, for some Def-
20 initelyTriviallySerializable type T.

21 *std::declval<SrcIter>() has the type global_ptr<T>, for the same
22 type T as with DestIter.

23 All source addresses must be global_ptr<T>'s referencing memory with
24 affinity to the same rank.

25 The length of the two sequences delimited by (src_runs_begin,
26 src_runs_end) and (dest_runs_begin, dest_runs_end) multiplied by
27 src_run_length and dest_run_length, respectively, must be the same.

28 This call has the same semantics as rget_irregular with the exception that,
29 for each sequence, all run lengths are the same and are factored out of the
30 sequences into two extra parameters src_run_length and dest_run_length,
31 which express the number of consecutive elements of type T in units of element
32 count. Thus, the iterated elements are no longer pairs, but just pointers.

33 The source sequence iterators must remain valid, and the underlying addresses
34 and memory contents must not be modified until operation completion is sig-
35 naled. Only after operation completion is signaled can the address sequences
36 and source memory be reclaimed by the application.

The destination sequence iterators must remain valid until operation completion is signaled.

Completions:

- *Operation:* Indicates completion of all aspects of the operation: the transfer and local stores are complete.

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.

UPC++ progress level: `internal`

14.2.6 Strided Put

```

13 template<std::size_t Dim, typename T,
14         typename Completions=decltype(operation_cx::as_future())>
15 RType rput_strided(
16     T const *src_base,
17     std::ptrdiff_t const *src_strides,
18     global_ptr<T> dest_base,
19     std::ptrdiff_t const *dest_strides,
20     std::size_t const *extents,
21     Completions cxs=Completions{});

```

```

23 template<std::size_t Dim, typename T,
24         typename Completions=decltype(operation_cx::as_future())>
25 RType rput_strided(
26     T const *src_base,
27     std::array<std::ptrdiff_t,Dim> const &src_strides,
28     global_ptr<T> dest_base,
29     std::array<std::ptrdiff_t,Dim> const &dest_strides,
30     std::array<std::size_t,Dim> const &extents,
31     Completions cxs=Completions{});

```

Precondition: T must be a DefinitelyTriviallySerializable type. All source addresses and destination global pointers must reference valid objects of type T. Each element `src_strides[i]`, `dest_strides[i]`, and `extents[i]` must be valid objects of their respective pointed-to type for all $0 \leq i < \text{Dim}$.

1 If `Dim == 0`, `src_strides`, `dest_strides`, and `extents` are ignored, and the
2 data movement performed is equivalent to `rput(src_base, dest_base, 1)`.

3 Otherwise, performs the semantic equivalent of many put's of type `T`. Let
4 the *index space* be the set of integer vectors of dimension `Dim` contained in
5 the bounding box with the inclusive lower bound at the all-zero origin, and
6 the exclusive upper bound equal to `extents`. For each index vector `index` in
7 this index space, a put will be executed with source and destination addresses
8 computed according to the following pseudo-code, where `dotprod` is the vector
9 dot product and pointer arithmetic is done in units of bytes (not elements of
10 \widehat{T}):

```
11 src_address = src_base + dotprod(index, src_strides)  
12 dest_address = dest_base + dotprod(index, dest_strides)
```

13 Note this implies the elements of the `src_strides` and `dest_strides` arrays
14 are expressed in units of bytes.

15 The destination memory regions must be completely disjoint and must not over-
16 lap with any source memory regions, otherwise behavior is undefined. Source
17 regions are permitted to overlap with each other.

18 The elements of type `T` residing in the source addresses must remain valid and
19 unmodified until source completion is signaled.

20 The contents of the `src_strides`, `dest_strides`, and `extents` arrays are con-
21 sumed synchronously before the call returns.

22 *Completions:*

- 23 • *Source:* Indicates that the source memory is no longer in use by UPC++
24 and may be reclaimed by the user.
- 25 • *Remote:* Indicates completion of the transfer of all values.
- 26 • *Operation:* Indicates completion of all aspects of the operation: the trans-
27 fer and remote stores are complete.

28 *C++ memory ordering:* The reads of the sources will have a *happens-before*
29 relationship with the source-completion notification actions (future readying,
30 promise fulfillment, or persona LPC enlistment). The writes to the destinations
31 will have a *happens-before* relationship with the operation-completion notifica-
32 tion actions (future readying, promise fulfillment, or persona LPC enlistment)
33 and remote-completion actions (RPC enlistment). For LPC and RPC com-
34 pletions, all evaluations *sequenced-before* this call will have a *happens-before*
35 relationship with the execution of the completion function.

36 *UPC++ progress level:* **internal**

1 **14.2.7 Strided Get**

```

2  template<std::size_t Dim, typename T,
3          typename Completions=decltype(operation_cx::as_future())>
4  RType rget_strided(
5      global_ptr<T> src_base,
6      std::ptrdiff_t const *src_strides,
7      T *dest_base,
8      std::ptrdiff_t const *dest_strides,
9      std::size_t const *extents,
10     Completions cxs=Completions{});
11
12  template<std::size_t Dim, typename T,
13          typename Completions=decltype(operation_cx::as_future())>
14  RType rget_strided(
15      global_ptr<T> src_base,
16      std::array<std::ptrdiff_t,Dim> const &src_strides,
17      T *dest_base,
18      std::array<std::ptrdiff_t,Dim> const &dest_strides,
19      std::array<std::size_t,Dim> const &extents,
20      Completions cxs=Completions{});

```

21 *Precondition:* T must be a DefinitelyTriviallySerializable type. All source global
22 pointers and destination addresses must reference valid objects of type T. Each
23 element `src_strides[i]`, `dest_strides[i]`, and `extents[i]` must be valid
24 objects of their respective pointed-to type for all $0 \leq i < \text{Dim}$.

25 If `Dim == 0`, `src_strides`, `dest_strides`, and `extents` are ignored, and the
26 data movement performed is equivalent to `rget(src_base, dest_base, 1)`.

27 Otherwise, performs the reverse direction of `rput_strided` where now the
28 source memory is remote and the destination is local.

29 The destination memory regions must be completely disjoint and must not over-
30 lap with any source memory regions, otherwise behavior is undefined. Source
31 regions are permitted to overlap with each other.

32 The elements of type T residing in the source addresses must remain valid and
33 unmodified until operation completion is signaled.

34 The contents of the `src_strides`, `dest_strides`, and `extents` arrays are con-
35 sumed synchronously before the call returns.

36 *Completions:*

- 1 • *Operation*: Indicates completion of all aspects of the operation: the trans-
2 fer and local stores are complete.

3 *C++ memory ordering*: The reads of the sources and writes to the destina-
4 tions will have a *happens-before* relationship with the operation-completion no-
5 tification actions (future readying, promise fulfillment, or persona LPC enlist-
6 ment). For LPC completions, all evaluations *sequenced-before* this call will have
7 a *happens-before* relationship with the execution of the completion function.

8 *UPC++ progress level*: **internal**

1 Chapter 15

2 Memory Kinds

3 The memory kinds interface enables the programmer to identify regions of memory requir-
4 ing different access methods or having different performance properties, and subsequently
5 rely on the UPC++ communication services to perform transfers among such regions (both
6 local and remote) in a manner transparent to the programmer. With GPU devices, HBM,
7 scratch-pad memories, NVRAM and various types of storage-class and fabric-attached
8 memory technologies featured in vendors' public road maps, UPC++ must be prepared to
9 deal efficiently with data transfers among all the memory technologies in any given system.

10 Since memory kinds will be implemented in Year 2, we defer detailed discussion until
11 next year.

1 Appendix A

2 Notes for Implementers

3 The following are possible implementations of template metaprogramming utilities for
4 UPC++ features.

5 A.1 future_element_t and future_element_moved_t

```
6 template<int I, typename T>
7 struct future_element; // undefined
8
9 template<int I, typename T, typename ...U>
10 struct future_element<I, future<T, U...>> {
11     typedef typename future_element<I-1, future<U...>>::type type;
12     typedef typename future_element<I-1, future<U...>>::moved_type
13         moved_type;
14 };
15
16 template<typename T, typename ...U>
17 struct future_element<0, future<T, U...>> {
18     typedef T type;
19     typedef T&& moved_type;
20 };
21
22 template<int I>
23 struct future_element<I, future<>> {
24     typedef void type;
25     typedef void moved_type;
26 };
27
```

```
1 template<int I, typename T>
2 using future_element_t = typename future_element<I, T>::type;
3
4 template<int I, typename T>
5 using future_element_moved_t =
6     typename future_element<I, T>::moved_type;
```

7 **A.2 future<T...>::when_all**

8 Utility types:

```
9 template<template<typename ...Us> class T, typename A, typename B>
10 struct concat_type; // undefined
11
12 template<template<typename ...Us> class T,
13         typename ...As, typename... Bs>
14 struct concat_type<T, T<As...>, T<Bs...> > {
15     typedef T<As..., Bs...> type;
16 };
17
18 template<template<typename ...Us> class T,
19         typename A, typename... Bs>
20 struct concat_element_types {
21     typedef typename concat_element_types<T, Bs...>::type rest;
22     typedef typename concat_type<T, A, rest>::type type;
23 };
24
25 template<template<typename ...Us> class T, typename A>
26 struct concat_element_types<T, A> {
27     typedef A type;
28 };
29
30 template<template<typename ...Us> class T, typename ...U>
31 using concat_element_types_t =
32     typename concat_element_types<T, U...>::type;
```

33 Declaration of future<T...>::when_all:

```
34 template<typename ...Futures>
35 concat_element_types_t<future, Futures...> when_all(Futures ...fs);
```

1 **A.3 to_future**

2 Utility types:

```
3 template<typename T>
4 struct future_type {
5     typedef future<T> type;
6 };
7
8 template<typename ...T>
9 struct future_type<future<T...>> {
10     typedef future<T...> type;
11 };
12
13 template<>
14 struct future_type<void> {
15     typedef future<> type;
16 };
17
18 template<typename T>
19 using future_type_t = typename future_type<T>::type;
```

20 Declaration of to_future:

```
21 template<typename T>
22 future_type_t<T> to_future(T future_or_value);
```

23 **A.4 future_invoke_result_t**

24 C++11-compliant implementation:

```
25 template<typename Func, typename... ArgTypes>
26 using future_invoke_result_t =
27     future_type_t<typename std::result_of<Func(ArgTypes...)>::type>;
```

28 C++17-compliant implementation:

```
29 template<typename Func, typename... ArgTypes>
30 using future_invoke_result_t =
31     future_type_t<std::invoke_result_t<Func, ArgTypes...>>;
```

Bibliography

- [1] ISO. *ISO/IEC 14882:2011(E) Information technology - Programming Languages - C++*. Geneva, Switzerland, 2012.
- [2] ISO. *ISO/IEC 14882:2014(E) Information technology - Programming Languages - C++, working draft*. Geneva, Switzerland, 2014.
- [3] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114, May 2014.

Index

- Affinity, 5
- allocate, 21
- atomic_domain, 78
 - add, 80
 - compare_exchange, 87
 - constructor, 78
 - dec, 85
 - default constructor, 78
 - destructor, 78
 - fetch_add, 81
 - fetch_dec, 86
 - fetch_inc, 84
 - fetch_sub, 82
 - inc, 83
 - load, 79
 - store, 79
 - sub, 82
- atomic_op, 78
- C++ Concepts, 5
- Collective, 5
- Conventions, 4
- CType, 52
 - operator |, 55
- current_persona, 73
- deallocate, 22
- default_persona, 73
- default_persona_scope, 75
- DefinitelySerializable, 5, 39
- DefinitelyTriviallySerializable, 5, 39
- delete_, 20
- delete_array, 21
- deserializing_iterator, 42, 44
- discharge, 67, 75
- dist_id, 100
 - here, 101
 - when_here, 101
- dist_object, 98
 - constructor, 98
 - destructor, 99
 - fetch, 100
 - id, 99
 - move constructor, 98
 - operator*, 100
 - operator->, 100
 - team, 99
 - variadic constructor, 98
- Execution Model, 3
- finalize, 9
- future, 32
 - default constructor, 33
 - destructor, 33
 - ready, 33
 - result, 33
 - result_moved, 34
 - result_tuple, 33
 - then, 34
 - wait, 34
- Futures and Promises, 5

-
- Global Pointer, 1, 6
 - global_ptr, 12
 - comparison operators, 17
 - comparison operators (STL specializations), 18
 - constructor (up-cast), 12
 - destructor, 13
 - is_local, 13
 - is_null, 13
 - local, 13
 - null constructor, 13
 - operator+, 14
 - operator++, 15, 16
 - operator+=", 14
 - operator-, 14, 15
 - operator--, 16
 - operator-=, 14
 - operator<<, 18
 - reinterpret_pointer_cast, 18
 - where, 14
 - Glossary, 5
 - init, 9
 - inrank_t, 12
 - is_definitely_serializable, 43
 - is_definitely_trivially_serializable, 43
 - liberate_master_persona, 73
 - Local, 6
 - make_future, 33
 - make_view, 42, 47
 - master_persona, 73
 - new_, 19
 - new_array, 20
 - Operation Completion, 6
 - operation_cx, 53
 - as_future, 53
 - as_lpc, 54
 - as_promise, 54
 - Persona, 6
 - persona, 74
 - default constructor, 71
 - destructor, 71
 - lpc, 72
 - lpc_ff, 72
 - persona_scope, 69
 - constructor, 74
 - constructor (with mutex), 74
 - destructor, 74
 - Private Object, 6
 - Progress, 6
 - progress, 66, 71
 - progress_level, 65, 71
 - progress_level::internal, 65
 - progress_level::none, 65
 - progress_level::user, 65
 - progress_required, 67, 75
 - promise, 35
 - default constructor, 35
 - destructor, 35
 - finalize, 37
 - fulfill_anonymous, 36
 - fulfill_result, 36
 - get_future, 36
 - require_anonymous, 36
 - Rank, 6
 - Referentially Transparent, 6
 - Remote, 6
 - Remote Procedure Call, 6
 - remote_cx, 53
 - as_rpc, 54
 - rget, 58
 - bulk rget, 58
 - rget_irregular, 105
 - rget_regular, 108

- `rget_strided`, 111
- `rpc`, 62
- `rpc_ff`, 61
- `rput`, 56
 - bulk `rput`, 57
- `rput_irregular`, 103
- `rput_regular`, 107
- `rput_strided`, 109
- `RType`, 52

- Serializable, 7, 39
- Serialization
 - Concepts, 39
 - Function objects, 41
 - Special cases, 41
 - User-defined, 38
 - View-based, 42
- `serialize`, 38
- Shared Segment, 7
- Source Completion, 7
- `source_cx`, 53
 - `as_blocking`, 54
 - `as_buffered`, 54
 - `as_future`, 53
 - `as_lpc`, 54
 - `as_promise`, 54

- Team, 7
- `team`, 89
 - `from_world`, 89
 - `rank_me`, 89
 - `rank_n`, 89
 - `split`, 90
- Thread, 7
- `to_future`, 35
- `top_persona_scope`, 74
- TriviallySerializable, 7, 39

- `view`, 42
 - default constructor, 46
 - `T*` iterator specialization, 46
 - with general iterator, 45
- `view_default_iterator_t`, 42, 44
- `when_all`, 35