UPC++ Specification
v1.0 Draft 4

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

September 27, 2017

Abstract

UPC++ is a C++11 library providing classes and functions that support Asynchronous Partitioned Global Address Space (APGAS) 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 to 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

Legal Disclaimer

Contents

Chapter 1

Overview and Scope

1.1 Preliminaries

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

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

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

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

Notably, global pointers are used in *one-sided* communication: bulk copying operations (RMA) similar to *memcpy* but across ranks (Ch. 7), and in Remote Procedure Calls

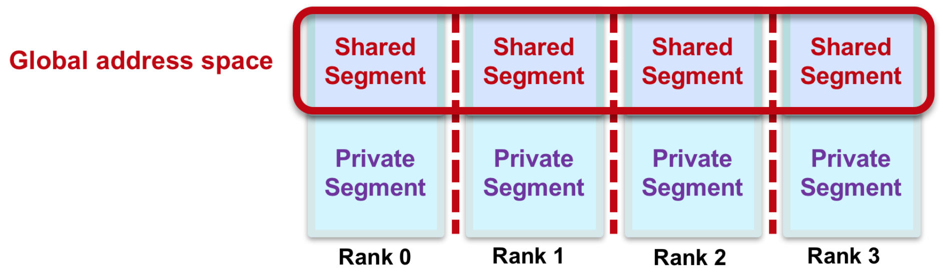

Figure 1.1: Abstract Machine Model of a PGAS program memory

(RPC, Ch. 8). RPC enables the programmer to ship functions to other ranks, which is useful in managing irregular distributed data structures. These ranks can push or pull data via global pointers. *Futures* and *Promises* (Ch. 5) are used to determine completion of communication or to provide handlers that respond to completion. Wherever possible, UPC++ will engage low-level hardware support for communication and this capability is crucial to UPC++'s support of *lightweight communication*.

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

UPC++ also avoids non-scalable constructs found in models such as UPC. For example, it does not support shared distributed arrays or shared scalars. Instead, it provides distributed objects, which can be used to similar ends (Ch. 12). Distributed objects are useful in solving the *bootstrapping problem*, whereby ranks need to distribute their local copies of global pointers to other ranks. Though UPC++ does not directly provide multidimensional arrays, applications that use UPC++ may define them. To this end, UPC++ supports non-contiguous data transfers: vector, indexed, and strided data (Ch. 13).

Because UPC++ does not provide separate concurrent threads to manage progress, UPC++ must manage all progress inside active calls to the library. UPC++ has been designed with a policy against the use of internal operating system threads. The strengths of this approach are improved user-visibility into the resource requirements of UPC++ and better interoperability with software packages and their possibly restrictive threading requirements. The consequence, however, is that the user must be conscientious to balance the need for making progress against the application's need for CPU cycles. Chapter 9 discusses subtleties of managing progress and how an application can arrange for UPC++ to advance the state of asynchronous communication.

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

UPC++ will support memory kinds (Ch. 14), whereby the programmer can identify regions of memory requiring different access methods or having different performance properties, such as device memory. Since memory kinds will be implemented in Year 2, we will defer their detailed discussion until next year.

1.2 Execution Model

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

UPC++ presents a *thread-aware* programming model. It assumes that only one thread of execution is interacting with any UPC++ object. The abstraction for thread-awareness in UPC++ is the *persona*. A `future` produced by a thread of execution is associated with its persona, and transferring the `future` to another thread must be accompanied by transferring the underlying persona. Each rank has a *master persona*, initially attached to the thread that calls `init`. Some UPC++ operations, such as `barrier`, require a thread to have exclusive access to the master persona to call them. Thus, the programmer is responsible for ensuring synchronized access to both personas and memory, and that access to shared 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 9 (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 7 and 8 describe the two forms of asynchronous one-sided communication, rput/rget and RPC, respectively. Chapter 9 discusses progress. Chapter 10 discusses atomic operations. Chapter 11 discusses teams, which are a means of organizing UPC++ ranks. Chapter 12 discusses distributed objects. Chapter 13 discusses non-contiguous data transfers. Chapter 14 discusses memory kinds.

1.5 Document 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 legal 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.

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

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

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

Operation completion. (7.2) The condition where a communication operation is complete with respect to the initiating rank, such that its effects are visible and that resources, such as source and destination memory regions, are no longer in use by `UPC++`.

Persona. (9.4) The abstraction for thread-awareness in `UPC++`. A `UPC++` persona object represents a collection of `UPC++`-internal state usually attributed to a single thread. By making it a proper construct, UPC++ allows a single OS thread to switch between multiple application-defined roles for processing notifications. Personas act as the receivers for notifications generated by the `UPC++` runtime.

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

Progress. (9) The means by which the application allows the `UPC++` runtime to advance the state of outstanding operations initiated by this or other ranks, to ensure they eventually complete.

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

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

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

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

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

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

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

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

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

Chapter 2

Init and Finalize

2.1 Overview

The `init` function must be called before any other `UPC++` function can be invoked. This can happen anywhere in the program, so long as it appears before any `UPC++` calls that require the library to be in an initialized state. The call is *collective*, meaning every process in the parallel job must enter this function if any are to participate in `UPC++` operations. While `init` can be called more than once by each process in a program, only the first invocation will initialize `UPC++`, and the rest will merely increment the internal count of how many times `init` has been called. For each `init` call, a matching `finalize` call must eventually be made. `init` and `finalize` are not re-entrant and must be called by only a single thread of execution in each process. The thread that calls `init` has the *master persona* attached to it (see section 9.5.1 for more details of threading behavior). After the number of calls to `finalize` matches the number of calls to `init`, no `UPC++` function that requires the library to be in an initialized state can be invoked until `UPC++` is reinitialized by a subsequent call to `init`.

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

2.2 Hello World

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

```
#include <upcxx/upcxx.hpp>
#include <iostream>
int main(int argc, char *argv[])
{
  upcxx::init();                        // initialize UPC++

  std::cout << "Hello World"
    << " ranks:" << upcxx::rank_n()     // how many UPC++ ranks?
    << " my rank: " << upcxx::rank_me() // which rank am I?
    << std::endl;

  upcxx::finalize();                    // finalize UPC++
  return 0;
}
```

Figure 2.1: *HelloWorld.cpp* program

different processes may even be interleaved.

2.3 API Reference

```
void init();
```

Preconditions: Called collectively by all processes in the parallel job. Calling thread must have the master persona (§9.5.1) if `UPC++` is in an already-initialized state.

If there have been no previous calls to `init`, or if all previous calls to `init` have had matching calls to `finalize`, then this routine initializes the `UPC++` library. Otherwise, leaves the library's state as is. Upon return, the calling thread will be attached to the master persona (§9.5.1).

This function is legal to call when `UPC++` *is in the uninitialized state.*

```
void finalize();
```

Preconditions: Called collectively by all processes in the parallel job. Calling thread must have the master persona (§9.5.1), and `UPC++` must be in an already-initialized state.

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

If this call uninitializes the `UPC++` library while there are any asynchronous operations still in-flight, behavior is undefined. An operation is defined as in-flight if it was initiated but still requires internal-level or user-level progress from any persona on any rank in the job before it can complete. It is left to the application to define and implement their own specific approach to ensuring quiescence of in-flight operations. A potential quiescence API is being considered for future versions and feedback is encouraged.

Chapter 3

Global Pointers

3.1 Overview

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

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

Global pointers have the following guarantees:

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

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

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

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

3.2 API Reference

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

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

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

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

It is illegal for `T` to have any cv qualifiers: `std::is_const<T>::value` and `std::is_volatile<T>::value` must both be false.

```
template<typename T>
struct global_ptr {
  using element_type = T;
  // ...
};
```

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

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

Precondition: `ptr` must be either null or an address in the shared segment (Ch. 4) of a rank in the local team (§11.2)

Constructs a global pointer corresponding to the given raw pointer. This constructor must be called explicitly.

UPC++ progress level: `none`

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

Constructs a global pointer corresponding to a null pointer.

This function is legal to call when `UPC++` *is in the uninitialized state.*

UPC++ progress level: `none`

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

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

This function is legal to call when `UPC++` *is in the uninitialized state.*

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

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

Precondition: `this->is_local()`

Converts this global pointer into a raw pointer.

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

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

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 greater than the current element, or a one-past-the-end pointer if the last element of the array is at `diff-1` positions greater than the current.

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

UPC++ progress level: `none`

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

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.

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

UPC++ progress level: `none`

```
template<typename T>
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`

```
template<typename T>
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`

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

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 copy of the original 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`

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

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

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`

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

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

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

```
template<typename T>
bool global_ptr<T>::operator==(global_ptr<T> rhs) const;
template<typename T>
bool global_ptr<T>::operator!=(global_ptr<T> rhs) const;
```

```
template<typename T>
bool global_ptr<T>::operator<(global_ptr<T> rhs) const;
template<typename T>
bool global_ptr<T>::operator<=(global_ptr<T> rhs) const;
template<typename T>
bool global_ptr<T>::operator>(global_ptr<T> rhs) const;
template<typename T>
bool global_ptr<T>::operator>=(global_ptr<T> rhs) const;
```

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

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

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

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

All other comparisons result in an unspecified value.

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

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

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

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

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

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

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

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

UPC++ progress level: `none`

Chapter 4

Storage Management

4.1 Overview

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

4.2 API Reference

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

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

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

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

UPC++ progress level: `none`

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

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

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

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

UPC++ progress level: `none`

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

Precondition: `T` must be DefaultConstructible.

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

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

UPC++ progress level: `none`

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

Precondition: `T` must be DefaultConstructible.

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

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

UPC++ progress level: `none`

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

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

Invokes the destructor on the given object and deallocates the storage allocated to it.

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

UPC++ progress level: `none`

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

Precondition: `T` must be Destructible. `g` must be a non-deallocated pointer that resulted from a call to `new_array<T>` on the current rank.

Invokes the destructor on each object in the given array and deallocates the storage allocated to it.

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

UPC++ progress level: `none`

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

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

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

UPC++ progress level: `none`

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

Precondition: `alignment` is a valid alignment.

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

UPC++ progress level: `none`

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

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

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

UPC++ progress level: `none`

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

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

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

UPC++ progress level: `none`

Chapter 5

Futures and Promises

5.1 Overview

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

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

5.2 The Basics of Asynchronous Communication

A programmer can invoke a non-blocking operation to be serviced by another rank, such as a one-sided get operation (Ch. 7) or a remote procedure call (Ch. 8). Such an operation

[1] Another mechanism, persona-targeted continuations, is discussed in §9.4.

creates an implicit promise and returns an associated future object to the user. When the operation completes, the future becomes ready, and it can be used to access the results. The following demonstrates an example using a remote get (see Ch. 9 on how to make progress with UPC++):

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

In general, a non-blocking operation will not complete immediately, so if a user needs to wait on the readiness of a future, they must do so in a loop. To facilitate this, we provide the `wait` member function, which waits on a future to complete while ensuring that sufficient progress (Ch. 9) is made on internal and user-level state:

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

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

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

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

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

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

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

5.3 Working with Promises

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

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

```
global_ptr<int> ptrs[10] = /* some remote pointers */;
// create a promise with no results
// the dependency count starts at one
promise<> prom;

// do 10 puts, registering each of them on the promise
for (int i = 0; i < 10; i++) {
  // rput implicitly registers itself on the given promise
  rput(ptrs[i], prom);
}

// fulfill initial anonymous dependency, since registration is done
prom.finalize_anonymous();

// wait for the rput operations to complete
prom.get_future().wait();
```

5.4 Advanced Callbacks

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

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

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

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

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

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

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

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

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

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

```
global_ptr<int> remote_array = /* some remote array */;

// retrieve remote_array[0]
future<int> elt0 = rget(remote_array);

// retrieve remote_array[remote_array[0]]
future<int> elt_indirect = elt0.then(
  [=](int index) {
    return rget(remote_array + index);
  }
);
```

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:

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

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:

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

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

A callback (made via `then`) can depend on multiple futures. We register the callback with a combined future, constructed with `when_all`. The `when_all` is restricted to combining lists of futures only. In the more general case, we may need to combine heterogeneous mixtures of future and non-future types. The `to_future` function template provides a further generalization, combining values from futures as well as raw (non-future) values themselves. While `when_all` can be used to meet this need (by wrapping raw values in calls to `make_future`), a call to `to_future` does this automatically:

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

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

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

```
template<typename ...T>
auto future<T...>::wait() {
  while (!ready()) {
    progress();
  }
  return result();
}
```

5.5 Execution Model

Futures have the capability to express dataflow/task-based programming, and other software frameworks provide thread-level parallelism by considering each callback to be a task that can be run in an arbitrary worker thread. This is not the case in `UPC++`. In order to maximize performance, our approach to futures is purposefully ambivalent to issues of concurrency. A `UPC++` implementation is allowed to take action as if the current thread is the only one that needs to be accounted for. This gives rise to a natural execution policy: callbacks registered against futures are always executed as soon as possible by the thread that discovers them. There are exactly two scenarios in which this may happen:

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

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

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

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

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

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

5.6 Anonymous Dependencies

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

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

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

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

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

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

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

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

While both empty and non-empty promises can be used to track anonymous dependencies, an empty promise is *only* able to track anonymous dependencies, so we expect that they will be the primary mechanism used to do so. As such, UPC++ operations that operate on promises make an important distinction between empty and non-empty promises: applying a UPC++ operation to an empty promise *does* increment its dependency count, calling a UPC++ operation on a non-empty promise *does not* increment its dependency count. The rationale for this is that an operation on a non-empty promise can only fulfill its initial, value-representing dependency, while an operation on an empty promise always fulfills an anonymous dependency. Rather than having the user manually increment the dependency count before calling an operation on an empty promise, UPC++ will implicitly perform this increment. This leads to the pattern, shown at the beginning of this chapter, of registering operations on an empty promise and then finalizing the promise to take it out of registration mode:

```
global_ptr<int> ptrs[10] = /* some remote pointers */;
promise<> prom; // dependency count is 1

for (int i = 0; i < 10; i++) {
  rput(ptrs[i], prom); // dependency count is incremented
} // dependency count is now 11

prom.finalize_anonymous(); // decrement count, making it 10

// wait for the 10 rput operations to complete
prom.get_future().wait();
```

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

5.7 Lifetime and Thread Safety

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

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

Given that `UPC++` futures and promises are already thread-unaware to allow the execution strategy to be straightforward and efficient, `UPC++` also makes no thread safety guarantees about internal state management. This enables creation of copies of a future to be a very cheap operation. For example, a future can be captured by value by a lambda function or passed by value without any performance penalties. On the other hand, the lack of thread safety means that sharing a future between threads must be handled with great caution. Even a simple operation such as making a copy of a future, as when passing it by value to a function, is unsafe if another thread is concurrently accessing an identical future, since the act of copying it can modify the internal management state. Thus, a mutex or other synchronization is required to ensure exclusive access to a future when performing any operation on it.

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

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

5.8 API Reference

UPC++ progress level for all functions in this chapter is: `none`

5.8.1 future

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

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

It is illegal for any type in `T` to be `void`.

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

Constructs a future that will never become ready.

This function is legal to call when `UPC++` *is in the uninitialized state.*

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

Destructs this future object.

This function is legal to call when `UPC++` *is in the uninitialized state.*

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

Constructs a trivially ready future from the given values.

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

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

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

Precondition: `this->ready()`

Retrieves the tuple of result values for this future.

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

Precondition: `this->ready()`

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

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

Precondition: `this->ready()`

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

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

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

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

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

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

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

Waits for the future by repeatedly attempting `UPC++` user-level progress and testing for readiness. See Ch. 9 for a discussion of progress. The return value is the same as that produced by calling `result()` on the future.

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

Given a variadic list of futures as arguments, constructs a future representing the readiness of all arguments. The results tuple of this future will be the concatenated results tuples of the arguments. The type parameters of the returned object (`CTypes...`) is the ordered concatenation of the type parameter lists of the types in `Futures`.

```
template<typename ...T>
future<CTypes...> to_future(T ...futures_or_results);
```

Given a variadic list of futures and/or non-futures as arguments, constructs a future representing the readiness of all the arguments that are futures. The results tuple of this future will be the concatenation of the result tuples of each future argument and the values of each non-future argument, in the order in which each argument occurs in `futures_or_results`. The type parameters of the returned object (`CTypes...`) is the concatenation of the type parameter lists of the future types in `T` and the non-future types themselves in `T`, in the order in which each type appears in `T`.

If none of the arguments are futures, then the resulting future object is trivially ready.

5.8.2 promise

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

C++ Concepts: DefaultConstructible, MoveConstructible, MoveAssignable, Destructible

It is illegal for any type in `T` to be `void`.

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

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

This function is legal to call when `UPC++` *is in the uninitialized state.*

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

Destructs this promise object.

This function is legal to call when `UPC++` *is in the uninitialized state.*

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

Precondition: The dependency count of this promise is greater than `(-count)` and greater than 0.

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

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

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

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

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

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

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

```
template<typename ...T>
void promise<T...>::finalize_anonymous();
```

Equivalent to `this->fulfill_anonymous(1)`.

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

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

Chapter 6

Serialization

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

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

1. Declare their type to be a friend of `access`

2. Implement the visitor function `serialize`

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

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

There are restrictions on which actions serialization/deserialization routines may perform. They are:

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

2. `UPC++` must perceive these routines as referentially transparent. Loosely, this means that the routines should be "pure" functions between the native representation and a flat sequence of bytes.

```
class UserType {
  // The user's fields and member declarations as usual.
  int member1, member2;
  // ...

  // To enable the serializer to visit the member fields,
  // the user provides this...
  friend class upcxx::access;

  // ...and this
  template<typename Archive>
  void serialize(Archive &ar, unsigned) {
    ar & this->member1;
    ar & this->member2;
    // ...
  }
};
```

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

3. The routines must be thread-safe and permit concurrent invocation from multiple threads, even when serializing the same object.

6.1 Functions

In §7.2 (*Completions*) and Chapter 8 (*Remote Procedure Calls*) there are several cases where a C++ *FunctionObject* is expected to execute on a destination rank. In these cases the function arguments are serialized as described in this chapter. The FunctionObject itself is converted to a function pointer offset from a known *sentinel* in the source program's *code segment*. The details of the implementation are not described here but typical allowed FunctionObjects are

- C functions
- C++ global and file-scope functions
- Class static functions
- lambda functions

Calling member functions on remote objects requires additional steps described in Chapter 12 (*Distributed Objects*).

Chapter 7

One-Sided Communication

7.1 Overview

The main one-sided communication functions for `UPC++` are `rput` and `rget`. Where possible, the underlying transport layer will use RDMA techniques to provide the lowest-latency transport possible. The type `T` used by `rput` or `rget` needs to be **Serializable**, either in the sense of C++ `TriviallyCopyable` or by overriding the global `upcxx::serialize` function as described in Chapter 6 (*Serialization*).

7.2 Completion

Memory movement operations come with the concept of completion, meaning that the effect of the operation is now visible and that resources, such as memory on the source and destination sides, are no longer in use by `UPC++`. The user has choices in how they would like `UPC++` to notify the application of completion: these are by future, promise, or continuation. Notification by future and promise was introduced in Ch. 5. Continuation style completion is explained in Ch. 9. An important aspect to clarify is that notification of completion only happens during user-level progress. Even if an operation completes early, including before the initiation operation returns, the application cannot learn this fact without entering user-progress. For futures and promises, only when the initiating thread (persona actually) enters user-level progress will the future or promise be eligible for taking on a readied or fulfilled state. Continuations will execute once a thread enters user-progress of the designated persona. See Ch. 9 for the full discussion on user-progress and personas.

7.3 API Reference

7.3.1 Remote Puts

```
template<typename T>
future<> rput(T value, global_ptr<T> dest);

template<typename T>
void rput(T value, global_ptr<T> dest, promise<> &completion);

template<typename T>
void rput(T value, global_ptr<T> dest,
          persona &completion_recipient,
          CompletionFunc completion_func);
```

Precondition: `T` must be Serializable. `dest` must reference a valid object of type `T`. In the second variant, `completion` must have a dependency count greater than zero. In the third variant, `CompletionFunc` must be a function-object type accepting no arguments, and the call `completion_func()` must not throw an exception.

Either serializes `value` immediately or copies it into an internal location for eventual serialization. After serialization, initiates a transfer of the data which will deserialize and store it in the memory referenced by `dest`.

Completion of the operation indicates that all aspects of the operation: serialization, deserialization, the remote store, and destruction of any internally managed `T` values are complete.

In the first variant, returns a future representing the completion of the operation.

In the second variant, the promise has its dependency count incremented immediately and fulfilled upon completion of the operation.

In the third variant, `completion_func` is enlisted in the given persona's user-progress upon completion of the operation (see §9.5.1 and Ch. 9).

C++ memory ordering: The writes to `dest` will have a *happens-before* relationship with the completion notification action (future readying, promise fulfillment, or persona continuation enlistment). In the third variant, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of `completion_func`.

UPC++ progress level: `internal`

```
template<typename T>
future<> rput(T const *src, global_ptr<T> dest, std::size_t count);

template<typename T>
void rput(T const *src, global_ptr<T> dest, std::size_t count
          promise<> &completion);

template<typename T, typename Func>
void rput(T const *src, global_ptr<T> dest, std::size_t count,
          persona &completion_recipient,
          CompletionFunc completion_func);
```

Precondition: `T` must be Serializable. Addresses in the intervals `[src,src+count)` and `[dest,dest+count)` must all reference valid objects of type `T`. No object may be referenced by both intervals. In the second variant, the `completion` promise must have a dependency count greater than zero. In the third variant, `CompletionFunc` must be a function-object type accepting no arguments, and the call `completion_func()` must not throw an exception.

Initiates an operation to serialize, transfer, deserialize, and store the `count` items of type `T` beginning at `src` to the memory beginning at `dest`.

Completion of this operation indicates that all source values have been serialized, deserialized, and the remote stores are complete. The values referenced in the `[src,src+count)` interval must not be modified until completion is indicated.

The first variant notifies completion via the readying of the returned future.

The second variant immediately increments the promise's dependency count and notifies completion by fulfilling that dependency.

The third variant notifies completion by enlisting `completion_func` in the given persona's user-progress (see §9.5.1 and Ch. 9).

C++ memory ordering: The writes to `dest` will have a *happens-before* relationship with the completion notification action (future readying, promise fulfillment, or persona continuation enlistment). In the third variant, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the execution of `completion_func`.

UPC++ progress level: `internal`

7.3.2 Remote Gets

```
template<typename T>
future<T> rget(global_ptr<T> src);

template<typename T>
void rget(global_ptr<T> src, promise<T> &completion);

template<typename T, typename CompletionFunc>
void rget(global_ptr<T> src,
          persona &completion_recipient,
          CompletionFunc completion_func);
```

Precondition: `T` must be Serializable. `src` must reference a valid object of type `T`. In the second variant, the `completion` promise must have a dependency count greater than zero and must not have had `fulfill_result` called on it before. In the third variant, `CompletionFunc` must be a function-object type accepting a single argument of type `T`, and `completion_func` must not throw an exception when invoked on its argument.

Initiates a transfer to this rank of a single value of type `T` located at `src`. Completion of the operation implies completion of serialization at the source side and deserialization at the initiator. Completion delivers the retrieved value directly in the notification.

The first variant notifies completion and the value by readying the future with that value.

The second variant notifies completion and the value by fulfilling the promise via `fulfill_result(value)`.

The third variant notifies completion and the value by enlisting the invocation of `completion_func(value)` in the given persona's user-progress (see §9.5.1 and Ch. 9).

C++ memory ordering: In the third variant, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the subsequent invocation of `completion_func(value)`.

UPC++ progress level: `internal`

```
template<typename T>
future<> rget(global_ptr<T> src, T *dest, std::size_t count);

template<typename T>
void rget(global_ptr<T> src, T *dest, std::size_t count,
          promise<> &completion);
```

```

template<typename T, typename CompletionFunc>
void rget(global_ptr<T> src, T *dest, std::size_t count,
          persona &completion_recipient,
          CompletionFunc completion_func);
```

Precondition: `T` must be Serializable. Addresses in the intervals `[src, src+count)` and `[dest, dest+count)` must all reference valid objects of type `T`. No object may be referenced by both intervals. In the second variant, `completion` must have a dependency count greater than zero. In the third variant, `CompletionFunc` must be a function-object type accepting no arguments, and the call `completion_func()` must not throw an exception.

Initiates a transfer of `count` values of type `T` beginning at `src` and assigns them to the locations beginning at `dest`.

Completion of the operation indicates completion of remote serialization, initiator-side deserialization, and all local assignments. The source values must not be modified until completion is notified.

The first variant notifies completion by readying the returned future.

The second variant immediately increments the promise's dependency count, and notifies completion by fulfilling that dependency.

The third variant notifies completion by enlisting `completion_func` to be invoked in the given persona's user-progress (see §9.5.1 and Ch. 9).

C++ memory ordering: In the third variant, all evaluations *sequenced-before* this call and the local assignments to `dest` will have a *happens-before* relationship with the invocation of `completion_func`.

UPC++ progress level: `internal`

Chapter 8

Remote Procedure Call

8.1 Overview

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

There are two main flavors of RPC in UPC++: *fire-and-forget* (rpc_ff) and *round trip* (rpc without the promise argument). Each takes a function Func together with variadic arguments Args.

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

The call rput_then_rpc combines a remote put with an rpc_ff, and the RPC is invoked after the remote put completes.

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

8.2 Remote Hello World Example

Figure 8.1 shows a simple alternative *Hello World* example where each rank issues an `rpc` to its neighbor, where the last rank wraps around to 0.

```
#include <upcxx/upcxx.hpp>
#include <iostream>
void hello_world(intrank_t num){
  std::cout << "Rank " << num <<"  told rank " << upcxx::rank_me()
     << " to say Hello World" << std::endl;
}
int main(int argc, char** argv[]){
  upcxx::init();           // Start UPC++ state
  intrank_t remote = (upcxx::rank_me()+1)%upcxx::rank_n();
  auto f = upcxx::rpc(remote, hello_world, upcxx::rank_me());
  f.wait();
  upcxx::finalize();       // Close down UPC++ state
  return 0;
}
```

Figure 8.1: HelloWorld with Remote Procedure Call

8.3 API Reference

```
template<typename Func, typename ...Args>
void rpc_ff(intrank_t recipient, Func &&func, Args &&...args);
```

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

The `func` and `args...` are serialized immediately and retained internally until they are eventually sent. After their receipt on `recipient`, they are deserialized and `func(args...)` is enlisted for execution during user-level progress of the master persona. So long as the sending persona continues to make internal-level progress it is guaranteed that the message will eventually arrive at the recipient. See §9.5.3 `progress_required` for an understanding of how much internal-progress is necessary.

Special handling is applied to those members of `args` which are either a reference to `dist_object` type (see §12 Distributed Objects) or a `team` (see §11

Teams). These are serialized by their `dist_id` or `team_id` respectively. The recipient deserializes the id's and waits asynchronously until all of them have a corresponding instance constructed on the recipient. When that occurs, `func` is called with the recipient's instance references in place of those supplied at the send site.

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

UPC++ progress level: `internal`

```
template<typename Func, typename ...Args>
future_invoke_result_t<Func, Args...>
  rpc(intrank_t recipient, Func &&func, Args &&...args);
```

Precondition: `Func` must be a Serializable type and a function-object type. Each of `Args...` must be either a Serializable type, or `dist_object<T>&`, or `team&`. Additionally, `std::result_of<Func(Args...)>::type` must be a Serializable type or `future<T...>`, where each type in `T...` must be Serializable. The call `func(args...)` must not throw an exception.

Similar to `rpc_ff`, this call sends `func` and `args...` to be executed remotely, but additionally returns a non-ready future which will be readied with the value returned from the remote invocation of `func(args...)`.

`func` and `args...` are either serialized immediately, or copy/moved (depending on the universal reference) to internal storage managed by `UPC++` and serialized sometime later (the returned future's readying indicates that serialization is complete). The serialized values are then sent to `recipient`, and upon receipt are deserialized and `func(args...)` is enlisted for execution during user-level progress of the master persona.

If the result of `func(args...)` is a future, the return type of `rpc` is the same as that of the result, and the recipient will wait for the future to become ready before sending its results back to the sender. Otherwise, the return type of `rpc` is a future that encapsulates the result of `func(args...)`, unless the result of `func` is `void`, in which case it is `future<>`. Within user-progress of the recipient's master persona, the result from invoking `func(args...)` will be immediately serialized and eventually sent back to the initiating rank. Upon receipt, it will be deserialized and the action of readying the final future with that value will be enlisted into user-progress of the initiating persona.

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

C++ memory ordering: All evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of `func`. The return from `func`, and readying of that return value if it is a future, will have a *happens-before* relationship with the readying of the final future.

UPC++ progress level: `internal`

```
template<typename Func, typename ...Args>
void rpc(intrank_t recipient,
         promise_invoke_result_t<Func, Args...> &pro,
         Func &&func, Args &&...args);
```

Precondition: `Func` must be a Serializable type and a function-object type. Each of `Args...` must be a Serializable type, or `dist_object<T>&`, or `team&`. Additionally, `std::result_of<Func(Args...)>::type` must be a Serializable type or `future<T...>`, where each type in `T...` must be Serializable. The call `func(args...)` must not throw an exception. The dependency count of `pro` must be greater than zero, and if it is a non-empty promise, then its non-anonymous dependency must not have been fulfilled.

Sends `func` and `args...` and sends back the result in the same way as the future-returning variant of `rpc`, but instead fulfills the given promise with the final value during user-progress of the initiating persona. If the result of `func(args...)` is of the form `future<T...>`, then `pro` must have the type `promise<T...>`. If the result is some other non-`void` type `T`, then `pro` must be of type `promise<T>`. And if the result is `void`, `pro` must be of type `promise<>`. In all cases where `pro` has type `promise<>`, the call to `rpc` increments the anonymous dependency count of `pro`.

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

C++ memory ordering: All evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of `func`. The return from `func`, and readying of that return value if it is a future, will have a *happens-before* relationship with the fulfillment of the promise.

UPC++ progress level: `internal`

```
template<typename T,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
future<> rput_then_rpc(
  T const *src, global_ptr<T> dest,
  std::size_t count,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);

template<typename T,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
void rput_then_rpc(
  T const *src, global_ptr<T> dest,
  std::size_t count,
  promise<> &source_completion,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);

template<typename T, typename SourceCompletionFunc,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
void rput_then_rpc(
  T const *src, global_ptr<T> dest,
  std::size_t count,
  persona &source_completion_persona,
  SourceCompletionFunc source_completion_func,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);
```

Precondition: `RemoteCompletionFunc` must be a Serializable type and a function-object type. Each of `RemoteCompletionArgs...` must either be a Serializable type, or `dist_object<U>&`, or `team&`. `SourceCompletionFunc` must be a function-object type. The calls `remote_completion_func(remote_completion_args...)` and `source_completion_func()` must not throw an exception. Either `dest` or `dest-1` must reference a valid object of type `T`. Addresses in the intervals `[src, src+count)` and `[dest,dest+count)` must all reference valid objects of type `T`. No object may be referenced by both intervals. For the second variant, the dependency count of `source_completion` must be greater than zero.

Initiates a transfer of `count` items of type `T` from the local memory at `src` to the memory referenced by `dest`. Sends `remote_completion_func` and `remote_completion_args...` to the rank `dest.where()` (in the same manner as `rpc_ff`) and enlists `remote_completion_func(remote_completion_args...)`

to be run in user-progress of the master persona after the transfer completes. Serialization of `remote_completion_func` and `remote_completion_args` happens during this function call.

The initiating rank is notified of source completion, which only indicates that serialization of the source memory has occurred and the contents can be reclaimed. Source completion does not indicate the puts have become visible or that `remote_completion_func` has run on the target rank.

In the first variant, the resulting future represents source completion of the transfer. In the second variant, the dependency count of the given promise is incremented, and a dependency is fulfilled upon source completion of the transfer. In the third variant, `source_completion_func` is enlisted to the given persona's user-progress upon source completion of the transfer. The memory referenced by `src` must not be modified until notification of source completion.

C++ memory ordering: All evaluations *sequenced-before* this call and the puts from this call will have a *happens-before* relationship with the invocation of `remote_completion_func`. In the third variant, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of `source_completion_func`.

UPC++ progress level: `internal`

Chapter 9

Progress

9.1 Overview

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

Progress in `UPC++` refers to how the calling application allows the `UPC++` internal runtime to advance the state of its outstanding asynchronous operations. Any asynchronous operation initiated by the user may require the application to give `UPC++` access to the execution thread periodically until the operation reports its completion. Such access is granted by simply making calls into `UPC++`. Each `UPC++` function's contract to the user contains its *progress guarantee* level. This is described by the members of the `upcxx::progress_level` enumerated type:

`progress_level::none` `UPC++` will not attempt to advance the progress of asynchronous operations.

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

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

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

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

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

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

9.2 Restricted Context

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

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

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

9.3 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.

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

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

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

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

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

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

```
namespace upcxx {

  struct persona_scope {
    // Make ‘p‘ the new current persona for this OS thread.
    persona_scope(persona &p);

    // Acquire ‘lock‘, then make ‘p‘ the new current persona for
    // this OS thread.
    template<typename Lock>
    persona_scope(Lock &lock, persona &p);

    // Pop ‘p‘ from persona stack, release ‘lock‘ if any.
    // Calling thread must be same for constructor and destructor.
    ~persona_scope();
  };

  persona_scope& top_persona_scope();

  persona_scope& default_persona_scope();

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

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

} // namespace upcxx
```

```
// Example demonstrating persona_scope.

upcxx::persona scheduler_persona;
std::mutex scheduler_lock;

{ // Scope block delimits domain of persona_scope instance.
  auto scope = upcxx::persona_scope(scheduler_lock, scheduler_persona);

  // All following upcxx actions will use ‘scheduler_persona‘
  // as current.

  // ...

  // ‘scope‘ destructs:
  // - ‘scheduler_persona‘ dropped from active set if it
  //   wasn't active before the scope's construction.
  // - Previously current persona revived.
  // - Lock released.
}
```

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.

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

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

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

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

In order to issue operations and concurrently retire them in a different thread, the user is strongly encouraged to use the callback-oriented API calls of UPC++ as opposed to the future or promise variants. An example of such a variant is:

```
template<typename T, typename CompletionFunc>
void upcxx::rput(T const *src, global_ptr<T> dest, std::size_t count,
                 persona &completion_recipient,
                 CompletionFunc completion_func);
```

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

9.5 API Reference

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

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

This call will always attempt to advance internal progress.

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

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

UPC++ progress level: `internal` or `user`

9.5.1 persona

```
class persona;
```

C++ Concepts: DefaultConstructible, Destructible

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

Constructs a persona object with no enqueued operations.

This function is legal to call when `UPC++` *is in the uninitialized state.*

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

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

This function is legal to call when `UPC++` *is in the uninitialized state.*

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

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

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

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

UPC++ progress level: `none`

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

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

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

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

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

```
void liberate_master_persona()
```

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

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

UPC++ progress level: `none`

9.5.2 persona_scope

```
class persona_scope;
```

C++ Concepts: Destructible, MoveConstructible

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

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

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

UPC++ progress level: `none`

```
template<typename Mutex>
persona_scope::persona_scope(Mutex &mutex, persona &p);
```

C++ Concepts of `Mutex`: Mutex

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

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

UPC++ progress level: `none`

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

Precondition: All `persona_scope`'s constructed on this thread since the construction of this instance have since destructed.

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

UPC++ progress level: `none`

```
persona_scope& top_persona_scope();
```

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

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

9.5.3 Outgoing Progress

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

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

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

- If `ps` *does not* include the master persona: A return value of `true` means that one or more of the personas indicated by `ps` requires further internal-progress to achieve completion of its outgoing operations. A value of `false` means that none of the personas indicated by `ps` require internal-progress, but internal-progress of the master persona might still be required.
- If `ps` *does* include the master persona: A return value of `true` means that one or more of the personas indicated by `ps` requires further internal-progress to achieve completion of its outgoing operations. A return value of `false` means that none of the non-master personas indicated by `ps` requires further internal-progress, but the master persona may or may not require further internal-progress.

UPC++ progress level: `none`

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

Advances internal-progress enough to ensure that `progress_required(ps)` returns `false`.

UPC++ progress level: `internal`

Chapter 10

Atomics

10.1 Overview

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

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

Each atomic operation works on a global pointer of an *approved atomic type*. Currently, the approved atomic types are a subset of fundamental integer types, specifically: `std::int32_t`, `std::uint32_t`, `std::int64_t`, and `std::uint64_t`. All atomic operations are non-blocking and return a future to indicate completion. UPC++ currently supports only a limited set of operations: get, put, and fetch-and-add.

10.2 API Reference

```
template<typename T>
```

```
future<T> atomic_get(global_ptr<T> p, std::memory_order order);
```

Precondition: `T` must be one of the approved atomic types. `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` or `std::memory_order_acquire`.

Initiates an atomic read of the object at location `p` and returns its value in a future.

C++ memory ordering: If `order` is `std::memory_order_acquire` then the read performed will have a *happens-before* relationship with the readying of the returned future and all evaluations *sequenced-after*.

UPC++ progress level: `internal`

```
template<typename T>
future<> atomic_put(global_ptr<T> p, T val,
                    std::memory_order order);
```

Precondition: `T` must be one of the approved atomic types. `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` or `std::memory_order_release`.

Initiates an atomic write of `val` to the location `p`. Completion of the write is indicated in the returned future.

C++ memory ordering: If `order` is `std::memory_order_release` then all evaluations *sequenced-before* this call will have a *happens-before* relationship with the write performed.

UPC++ progress level: `internal`

```
template<typename T>
future<T> atomic_fetch_add(global_ptr<T> p, T val,
                           std::memory_order order);
```

Precondition: `T` must be one of the approved atomic types. `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`.

Initiates the atomic read-modify-write operation consisting of: reading the value of the object located at `p`, adding `val` to it, and writing the new value back. The value returned in the future is the one initially read.

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 readying of the returned future and all evaluations *sequenced-after*.

UPC++ progress level: `internal`

Chapter 11

Teams

11.1 Overview

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

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

11.2 Local Teams

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

11.3 API Reference

11.3.1 team

```
class team;
```

C++ Concepts: MoveConstructible, Destructible

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

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

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

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

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

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

UPC++ progress level: unspecified between `none` and `internal`

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

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

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

UPC++ progress level: unspecified between `none` and `internal`

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

Precondition: This function must be called collectively by all the ranks in this team, and it must be called by the thread that has the master persona (§9.5.1). No two ranks in the collective call may specify the same combination of `color` and `key`.

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

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

UPC++ progress level: `user`

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

Precondition: Calling thread must have the master persona.

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

UPC++ progress level: `none`

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

Precondition: Calling thread must have the master persona.

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

UPC++ progress level: `none`

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

Returns the universal name associated with this team.

UPC++ progress level: `none`

11.3.2 team_id

```
class team_id;
```

C++ Concepts: PODType, EqualityComparable, LessThanComparable, hashable

A universal name representing a team.

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

Precondition: The current rank must be a member of the `team` associated with this name, and it must have completed creation of the `team`.

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

UPC++ progress level: `none`

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

Precondition: The current rank must be a member of the `team` associated with this name. The calling thread must have the master persona.

Retrieves a future representing when the current rank constructs the `team` corresponding to this name.

UPC++ progress level: `none`

11.3.3 Fundamental Teams

```
team& world();
```

Returns a reference to the team representing all the ranks in the program. It is illegal to perform a move on the returned team.

UPC++ progress level: `none`

```
intrank_t rank_n();
```

Returns the number of ranks that are in the world team. Equivalent to `world().rank_n()`.

UPC++ progress level: `none`

```
intrank_t rank_me();
```

Returns the peer index of the caller in the world team. Equivalent to `world().rank_me()`.

UPC++ progress level: `none`

```
team& local_team();
```

Returns a reference to the local team containing this rank. A local team represents a set of ranks which share physical memory (§11.2). It is illegal to perform a move on the returned team.

UPC++ progress level: `none`

```
bool local_team_contains(intrank_t world_index);
```

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

Determines if `world_index` is a member of the local team containing the this rank (§11.2). Equivalent to: `local_team().from_world(world_index,-1) >= 0`

UPC++ progress level: `none`

11.3.4 Collectives

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

Precondition: This function must be called collectively by all the ranks in the given team, and it must be called by the thread that has the master persona (§9.5.1).

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

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

UPC++ progress level: `user`

```
future<> barrier_async(team &team = world());
```

Precondition: This function must be called collectively by all the ranks in the given team, and it must be called by the thread that has the master persona (§9.5.1).

Initiates an asynchronous barrier operation over the given team. The call will return without waiting for other ranks to make the call. The returned future will only become ready after all other ranks in the team have entered the call.

C++ memory ordering: With respect to all threads participating in this collective, all evaluations which are *sequenced-before* their respective thread's invocation of this call will have a *happens-before* relationship with all evaluations sequenced after the signaling of the returned futures.

UPC++ progress level: `internal`

```
template<typename T, typename BinaryOp>
future<T> allreduce(T &&value, BinaryOp &&op, team &team = world());
```

Precondition: This function must be called collectively by all the ranks in the given team, and it must be called by the thread that has the master persona (§9.5.1). `T` must be Serializable. `BinaryOp` must be a function-object type representing an associative and commutative mathematical operation taking two values of type `T` and returning a value implicitly convertible to `T`. `BinaryOp` must be referentially transparent and concurrently invocable. `BinaryOp` may not invoke any `UPC++` routine with a progress level other than none.

Performs a reduction operation over the ranks in the given team. If the team contains only a single rank, then the resulting future will hold `value`. Otherwise, initiates an asynchronous reduction over the values provided by each rank. The reduction is performed in some non-deterministic order by applying `op` to combine values and intermediate results. Each rank receives the result of the reduction in the returned future.

C++ memory ordering: With respect to all threads participating in this collective, all evaluations which are *sequenced-before* their respective thread's invocation of this call will have a *happens-before* relationship with all evaluations sequenced after the signaling of the returned futures.

UPC++ progress level: `internal`

```
template<typename T>
future<T> broadcast(T &&value, intrank_t sender,
                    team &team = world());
```

```
template<typename T>
future<> broadcast(T *buffer, std::size_t count,
                   intrank_t sender, team &team = world());
```

Precondition: The function must be called collectively by the ranks in the given team, and it must be called by the thread that has the master persona (§9.5.1). The value of `sender`, and `count` in the second variant, must be the same across all callers. In the second variant, the addresses in the interval `[buffer,buffer+count)` must all reference valid objects of type `T`. The type `T` must be Serializable.

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

C++ memory ordering: With respect to all threads participating in this collective, all evaluations which are *sequenced-before* the producing thread's invocation of this call will have a *happens-before* relationship with all evaluations sequenced after the signaling of the returned futures.

UPC++ progress level: `internal`

Chapter 12

Distributed Objects

12.1 Overview

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

1. Universal distributed object naming: per-object names that can be transmitted to other ranks while retaining their meaning.

2. Name-to-this mapping: Mapping between the universal name and the current rank's memory address holding that distributed object's state for the rank (the current rank's `this` pointer).

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

12.2 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 trivially Serializable. It has the members `.here()` and `.when_here()` for retrieving the resident `dist_object<T>` instance registered with the name.

12.3 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. 8).

12.4 API Reference

```
template<typename T>
struct dist_object<T>;
```

C++ Concepts: MoveConstructible, Destructible

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

Precondition: Calling thread must have the master persona.

Constructs this rank's member of the distributed object identified by the collective calling context across `team`. The initial value for this rank is given in `value`. The future returned from `dist_id<T>::when_here` for the corresponding `dist_id<T>` will be readied during this constructor. This implies that continuations waiting for that future will execute before the constructor returns.

UPC++ progress level: `none`

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

Precondition: Calling thread must have the master persona.

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

UPC++ progress level: `none`

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

Precondition: Calling thread must have the master persona.

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

UPC++ progress level: `none`

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

Precondition: Calling thread must have the master persona.

If this instance has not been invalidated by being passed to the move constructor, then this will destroy the current rank's member of the distributed object. `~T()` will be invoked on the resident instance, and further lookups on this rank using the `dist_id<T>` corresponding to this distributed object will have undefined behavior. If this instance has been invalidated by a move, then this call will have no effect.

UPC++ progress level: `none`

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

Returns the `dist_id<T>` representing the universal name of this distributed object.

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

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

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

UPC++ progress level: `none`

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

C++ Concepts: PODType, EqualityComparable, LessThanComparable, hashable

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

Precondition: The current rank's `dist_object<T>` instance associated with this name must not have been destroyed. The calling thread must have the master persona.

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

UPC++ progress level: `none`

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

Precondition: The current rank's `dist_object<T>` instance associated with this name must be alive. The calling thread must have the master persona.

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

UPC++ progress level: `none`

Chapter 13

Non-Contiguous One-Sided Communication

13.1 Overview

UPC++ provides functions to perform one-sided communications similar to rget and rput which are dedicated to handle data stored in non-contiguous buffers.

These functions are denoted with the `fragmented` keyword, and take two sequences of `std::pair` (or more generally `std::tuple`) describing how source and destination fragmented buffers should be accessed.

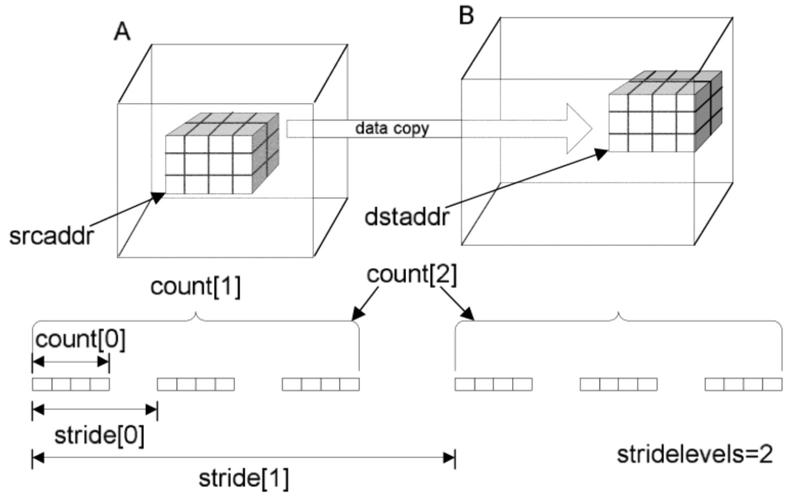

Figure 13.1: An example of a unit-stride *i* transfer between a `src` address and a `dst` address

The most general version of the API requires each `std::pair` to contain a local or global pointer to a memory location in the first member while the second member contains the size of the contiguous chunk of memory to be transferred.

A second set of functions targets identical chunk sizes, thus requiring the user to provide pointers only. These functions are denoted by the `regular` keyword.

Finally, the third set of functions provide an API for strided accesses starting from two given source and destination addresses. An example of such a transfer is depicted in Figure 13.1. These are denoted by the `strided` keyword.

Each of the functions also has a `then_rpc` variant which executes a remote procedure call targeting the destination rank to signal completion of the transfer.

13.2 API Reference

13.2.1 Fragmented Put

```
// future variant
template<typename SrcIter, typename DestIter>
future<> rput_fragmented(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  DestIter dest_runs_begin, DestIter dest_runs_end);

// promise variant
template<typename SrcIter, typename DestIter>
void rput_fragmented(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  promise<> &completion);

// continuation variant
template<typename SrcIter, typename DestIter,
         typename CompletionFunc>
void rput_fragmented(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  persona &completion_recipient,
  CompletionFunc completion_func);
```

Preconditions:

`SrcIter` and `DestIter` both satisfy the ForwardIterator C++ concept.

`std::get<0>(*std::declval<SrcIter>())` has a return type convertible to `T const*`, for some type `T`.

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

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

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

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

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

`CompletionFunc` is a function-object type.

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

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

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

The sequence iterators must remain valid, and the underlying addresses and source memory contents must stay constant until completion is signaled. Only after completion is signaled can the address sequences and source memory be reclaimed by the application.

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

In the future variant, completion is returned as a future.

In the promise variant, an anonymous dependency is added to the promise during the call and is fulfilled upon completion.

In the continuation variant, the `completion_func` function object is submitted to `completion_recipient`'s user-progress continuation queue upon completion.

C++ memory ordering: In the continuation variant, all evaluations *sequenced-before* this call and the put operations from this call will have a *happens-before* relationship with the invocation of `completion_func`.

UPC++ progress level: `internal`

13.2.2 Fragmented Get

```
// future variant
template<typename SrcIter, typename DestIter>
future<> rget_fragmented(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  DestIter dest_runs_begin, DestIter dest_runs_end);

// promise variant
template<typename SrcIter, typename DestIter>
void rget_fragmented(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  promise<> &completion);

// continuation variant
template<typename SrcIter, typename DestIter,
         typename CompletionFunc>
void rget_fragmented(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  persona &completion_recipient,
  CompletionFunc completion_func);
```

Preconditions:

`SrcIter` and `DestIter` both satisfy the ForwardIterator C++ concept.

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

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

`std::get<0>(*std::declval<DestIter>())` has the type `T*`, for some type `T`.

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

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

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

`CompletionFunc` is a function-object type.

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

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

As an example of valid types for individual runs, `DestIter` could be an iterator over elements of type `std::pair<T*, std::size_t>`, and `SrcIter` an iterator over `std::pair<global_ptr<T>, std::size_t>`. Variations replacing `std::pair` with `std::tuple` or `size_t` with other primitive integral types are also valid.

The sequence iterators must remain valid, and the underlying addresses and source memory contents must stay constant until completion is signaled. Only after completion is signaled can the address sequences and source memory be reclaimed by the application.

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

In the future variant, completion is returned as a future.

In the promise variant, an anonymous dependency is added to the promise during the call and is fulfilled upon completion.

In the continuation variant, the `completion_func` function object is submitted to `completion_recipient`'s user-progress continuation queue upon completion.

C++ memory ordering: In the continuation variant, all evaluations *sequenced-before* this call and the gets from this call will have a *happens-before* relationship with the invocation of `completion_func`.

UPC++ progress level: `internal`

13.2.3 Fragmented Put then RPC

```
// future variant
template<typename SrcIter, typename DestIter,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
future<> rput_fragmented_then_rpc(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  intrank_t recipient,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);

// promise variant
template<typename SrcIter, typename DestIter,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
void rput_fragmented_then_rpc(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  promise<> &source_completion,
  intrank_t recipient,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);

// continuation variant
template<typename SrcIter, typename DestIter,
         typename SourceCompletionFunc,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
void rput_fragmented_then_rpc(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  DestIter dest_runs_begin, DestIter dest_runs_end,

  persona &source_completion_recipient,
  SourceCompletionFunc source_completion_func,

  intrank_t recipient,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);
```

Preconditions: Same as those in `rput_fragmented` with the addition that `RemoteCompletionFunc` be Serializable and a function-object type and that all `RemoteCompletionArgs` be Serializable, or `dist_object<U>&`, or `team&`. The calls `remote_completion_func(remote_completion_args...)`

and `source_completion_func()` must not throw an exception. All remote memory referenced by the destination address sequence has affinity with rank `recipient`.

Performs the same series of puts as in `rput_fragmented`. Completion of the operation triggers a `rpc_ff` consisting of `remote_completion_func` invoked against `remote_completion_args` to the `recipient` rank. The current rank does not have access to this completion event. Instead, the current rank is notified of source completion. Source completion indicates only that the source and destination memory address sequences and source memory contents can be reclaimed. Source completion does not indicate the puts have become visible.

Serialization of `remote_completion_func` and `remote_completion_args` happen during the function call.

In the future variant, source completion is returned as a future.

In the promise variant, an anonymous dependency is added to the promise during the call and is fulfilled upon source completion.

In the continuation variant, the `source_completion_func` function object is submitted to `source_completion_recipient`'s user-progress continuation queue upon source completion.

C++ memory ordering: All evaluations *sequenced-before* this call and the puts from this call will have a *happens-before* relationship with the invocation of `remote_completion_func`. In the continuation variant, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of `source_completion_func`.

UPC++ progress level: `internal`

13.2.4 Fragmented Regular Put

```
// future variant
template<typename SrcIter, typename DestIter>
future<> rput_fragmented_regular(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  std::size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  std::size_t dest_run_length);

// promise variant
template<typename SrcIter, typename DestIter>
void rput_fragmented_regular(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  std::size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  std::size_t dest_run_length,
  promise<> &completion);

// continuation variant
template<typename SrcIter, typename DestIter,
         typename CompletionFunc>
void rput_fragmented_regular(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  std::size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  std::size_t dest_run_length,
  persona &completion_recipient,
  CompletionFunc completion_func);
```

Preconditions:

`SrcIter` and `DestIter` both satisfy the ForwardIterator C++ concept.

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

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

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

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

`CompletionFunc` is a function-object type.

These calls have the same semantics as their `rput_fragmented` counterparts with the difference that, for each sequence, all run lengths are the same and are factored out of the sequences into two extra parameters `src_run_length` and `dest_run_length`. Thus the iterated elements are no longer pairs, but just pointers (the first pair component).

The sequence iterators must remain valid, and the underlying addresses and source memory contents must stay constant until completion is signaled. Only after completion is signaled can the address sequences and source memory be reclaimed by the application.

In the future variant, completion is returned as a future.

In the promise variant, an anonymous dependency is added to the promise during the call and is fulfilled upon completion.

In the continuation variant, the `completion_func` function object is submitted to `completion_recipient`'s user-progress continuation queue upon completion.

C++ memory ordering: In the continuation variant, all evaluations *sequenced-before* this call and the puts from this call will have a *happens-before* relationship with the invocation of `completion_func`.

UPC++ progress level: `internal`

13.2.5 Fragmented Regular Get

```
// future variant
template<typename SrcIter, typename DestIter>
future<> rget_fragmented_regular(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  std::size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  std::size_t dest_run_length);

// promise variant
template<typename SrcIter, typename DestIter>
void rget_fragmented_regular(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  std::size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  std::size_t dest_run_length,
  promise<> &completion);

// continuation variant
template<typename SrcIter, typename DestIter,
         typename CompletionFunc>
void rget_fragmented_regular(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  std::size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  std::size_t dest_run_length,
  persona &completion_recipient,
  CompletionFunc completion_func);
```

Preconditions:

`SrcIter` and `DestIter` both satisfy the ForwardIterator C++ concept.

`*std::declval<DestIter>()` has a type convertible to `T*`, for some type `T`.

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

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

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

`CompletionFunc` is a function-object type.

These calls have the same semantics as their `rget_fragmented` counterparts with the difference that, for both sequences, all run lengths are the same and are factored out of the sequences into two extra parameters `src_run_length` and `dest_run_length`. Thus the iterated elements are no longer pairs, but just pointers (the first component).

The sequence iterators must remain valid, and the underlying addresses and source memory contents must stay constant until completion is signaled. Only after completion is signaled can the address sequences and source memory be reclaimed by the application.

In the future variant, completion is returned as a future.

In the promise variant, an anonymous dependency is added to the promise during the call and is fulfilled upon completion.

In the continuation variant, the `completion_func` function object is submitted to `completion_recipient`'s user-progress continuation queue upon completion.

C++ memory ordering: In the continuation variant, all evaluations *sequenced-before* this call and the gets from this call will have a *happens-before* relationship with the invocation of `completion_func`.

UPC++ progress level: `internal`

13.2.6 Fragmented Regular Put then RPC

```
// future variant
template<typename SrcIter, typename DestIter>
future<> rput_fragmented_regular_then_rpc(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  std::size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  std::size_t dest_run_length,
  intrank_t recipient,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);

// promise variant
template<typename SrcIter, typename DestIter,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
void rput_fragmented_regular_then_rpc(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  std::size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  std::size_t dest_run_length,
  intrank_t recipient,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);

// continuation variant
template<typename SrcIter, typename DestIter,
         typename SourceCompletionFunc,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
void rput_fragmented_regular_then_rpc(
  SrcIter src_runs_begin, SrcIter src_runs_end,
  std::size_t src_run_length,
  DestIter dest_runs_begin, DestIter dest_runs_end,
  std::size_t dest_run_length,

  persona &source_completion_recipient,
  SourceCompletionFunc source_completion_func,

  intrank_t recipient,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);
```

Preconditions: Same as those in `rput_fragmented_regular` with the addition that `RemoteCompletionFunc` be Serializable and a function-object type and that all `RemoteCompletionArgs` be Serializable, or `dist_object<U>&`, or `team&`. The calls `remote_completion_func(remote_completion_args...)`

and `source_completion_func()` must not throw an exception. All memory referenced in the destination address sequence must have affinity with the `recipient` rank.

Performs the same series of puts as in `rput_fragmented_regular`. Completion of the operation triggers a `rpc_ff` consisting of `remote_completion_func` invoked against `remote_completion_args` to the `recipient` rank. The current rank does not have access to this completion event. Instead, the current rank is notified of source completion. Source completion indicates only that the source and destination memory address sequences and source memory contents can be reclaimed. Source completion does not indicate the puts have become visible.

Serialization of `remote_completion_func` and `remote_completion_args` happen during the function call.

In the future variant, source completion is returned as a future.

In the promise variant, an anonymous dependency is added to the promise during the call and is fulfilled upon source completion.

In the continuation variant, the `source_completion_func` function object is submitted to `source_completion_recipient`'s user-progress continuation queue upon source completion.

C++ memory ordering: All evaluations *sequenced-before* this call and the puts from this call will have a *happens-before* relationship with the invocation of `remote_completion_func`. In the continuation variant, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of `source_completion_func`.

UPC++ progress level: `internal`

13.2.7 Strided Put

```
// future variant
template<typename T, int Dim>
future<> rput_strided(
  T const *src_base,
  std::ptrdiff_t const *src_strides,
  global_ptr<T> dest_base,
  std::ptrdiff_t const *dest_strides,
  std::size_t const *extents);

// promise variant
template<typename T, int Dim>
```

```
void rput_strided(
  T const *src_base,
  std::ptrdiff_t const *src_strides,
  global_ptr<T> dest_base,
  std::ptrdiff_t const *dest_strides,
  std::size_t const *extents,
  promise<> &completion);

// continuation variant
template<typename T, int Dim, typename CompletionFunc>
void rput_strided(
  T const *src_base,
  std::ptrdiff_t const *src_strides,
  global_ptr<T> dest_base,
  std::ptrdiff_t const *dest_strides,
  std::size_t const *extents,
  persona &completion_recipient,
  CompletionFunc completion_func);
```

Precondition: `T` must be a Serializable type. `Dim` must be non-negative. All source addresses and destination global pointers must reference valid objects of type `T`. Each of `src_strides[i]`, `dest_strides[i]`, and `extents[i]` must be valid objects of their respective pointed-to type for all `0 <= i < Dim`.

If Dim == 0, `src_strides`, `dest_strides`, and `extents` are ignored, and the data movement performed is equivalent to `rput(src_base, dest_base, 1)`.

Otherwise, performs the semantic equivalent of many put's of type `T`. Let the *index space* be the set of integer vectors of dimension `Dim` in the bounding box with the inclusive lower bound at the all-zero origin, and the exclusive upper bound equal to `extents`. For each index vector `index` in the index space, there will be a put with source and destination addresses computed as:

```
// "dot" is the vector dot product.
// Pointer arithmetic is done in bytes, not elements of T.
// "dest_base" is a global_ptr, following syntax is
// pseudo-code.
src_address = src_base + dot(index, src_strides)
dest_address = dest_base + dot(index, dest_strides)
```

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

The contents of the source addresses must remain valid and constant until completion is signaled.

In the future variant, completion is returned as a future.

In the promise variant, an anonymous dependency is added to the promise during the call and is fulfilled upon completion.

In the continuation variant, the `completion_func` function object is submitted to `completion_recipient`'s user-progress continuation queue upon completion.

C++ memory ordering: In the continuation variant, all evaluations *sequenced-before* this call and the puts from this call will have a *happens-before* relationship with the invocation of `completion_func`.

UPC++ progress level: `internal`

13.2.8 Strided Get

```
// future variant
template<typename T, int Dim>
future<> rget_strided(
  global_ptr<T> src_base,
  std::ptrdiff_t const *src_strides,
  T *dest_base,
  std::ptrdiff_t const *dest_strides,
  std::size_t const *extents);

// promise variant
template<typename T, int Dim>
void rget_strided(
  global_ptr<T> src_base,
  std::ptrdiff_t const *src_strides,
  T *dest_base,
  std::ptrdiff_t const *dest_strides,
  std::size_t const *extents,
  promise<> &completion);

// continuation variant
template<typename T, int Dim, typename CompletionFunc>
void rget_strided(
  global_ptr<T> src_base,
  std::ptrdiff_t const *src_strides,
  T *dest_base,
  std::ptrdiff_t const *dest_strides,
  std::size_t const *extents,
  persona &completion_recipient,
  CompletionFunc completion_func);
```

Precondition: `T` must be a Serializable type. `Dim` must be non-negative. All source global pointers and destination addresses must reference valid objects of type `T`. Each of `src_strides[i]`, `dest_strides[i]`, and `extents[i]` must be valid objects of their respective pointed-to type for all `0 <= i < Dim`.

If Dim == 0, `src_strides`, `dest_strides`, and `extents` are ignored, and the data movement performed is equivalent to `rget(src_base, dest_base, 1)`.

Otherwise, performs the reverse direction of `rput_strided` where now the source memory is remote and the destination is local.

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

The contents of the source addresses must remain valid and constant until completion is signaled.

In the future variant, completion is returned as a future.

In the promise variant, an anonymous dependency is added to the promise during the call and is fulfilled upon completion.

In the continuation variant, the `completion_func` function object is submitted to `completion_recipient`'s user-progress continuation queue upon completion.

C++ memory ordering: In the continuation variant, all evaluations *sequenced-before* this call and the gets from this call will have a *happens-before* relationship with the invocation of `completion_func`.

UPC++ progress level: `internal`

13.2.9 Strided Put then RPC

```
// future variant
template<typename T, int Dim,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
future<> rput_strided_then_rpc(
  T const *src_base,
  std::ptrdiff_t const *src_strides,
  global_ptr<T> dest_base,
  std::ptrdiff_t const *dest_strides,
  std::size_t const *extents,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);

// promise variant
template<typename T, int Dim,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
void rput_strided_then_rpc(
  T const *src_base,
  std::ptrdiff_t const *src_strides,
  global_ptr<T> dest_base,
  std::ptrdiff_t const *dest_strides,
  std::size_t const *extents,
  promise<> &source_completion,
  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);

// continuation variant
template<typename T, int Dim,
         typename SourceCompletionFunc,
         typename RemoteCompletionFunc,
         typename ...RemoteCompletionArgs>
void rput_strided_then_rpc(
  T const *src_base,
  std::ptrdiff_t const *src_strides,
  global_ptr<T> dest_base,
  std::ptrdiff_t const *dest_strides,
  std::size_t const *extents,

  persona &source_completion_recipient,
  SourceCompletionFunc source_completion_func,

  RemoteCompletionFunc &&remote_completion_func,
  RemoteCompletionArgs &&...remote_completion_args);
```

Preconditions: Same as those in `rput_strided` with the addition that `RemoteCompletionFunc` be Serializable and a function-object type and that all `RemoteCompletionArgs` be Serializable, or `dist_object<U>&`, or `team&`. The calls `remote_completion_func(remote_completion_args...)`

and `source_completion_func()` must not throw an exception. Either `dest_base` or `dest_base-1` must reference a valid object of type `T`.

Performs the same series of puts as `rput_strided` except for the completion semantics. Upon completion of the puts, the rank `dest_base.where()` is delivered an `rpc_ff` of `remote_completion_func` invoked against `remote_completion_args`. Source completion is returned to the caller. Source completion signals that memory referenced by the source addresses may now be modified or reclaimed, it does not indicate completion of the puts.

Serialization of `remote_completion_func` and `remote_completion_args` occur during this call.

In the future variant, source completion is returned as a future.

In the promise variant, an anonymous dependency is added to the promise during the call and is fulfilled upon source completion.

In the continuation variant, the `source_completion_func` function object is submitted to `source_completion_recipient`'s user-progress continuation queue upon source completion.

C++ memory ordering: All evaluations *sequenced-before* this call and the puts from this call will have a *happens-before* relationship with the invocation of `remote_completion_func`. In the continuation variant, all evaluations *sequenced-before* this call will have a *happens-before* relationship with the invocation of `source_completion_func`.

UPC++ progress level: `internal`

Chapter 14

Memory Kinds

The memory kinds interface enables the programmer to identify regions of memory requiring different access methods or having different performance properties, and subsequently rely on the UPC++ communication services to perform transfers among such regions (both local and remote) in a manner transparent to the programmer. With GPU devices, HBM, scratch-pad memories, NVRAM and various types of storage-class and fabric-attached memory technologies featured in vendors' public road maps, UPC++ must be prepared to deal efficiently with data transfers among all the memory technologies in any given system.

Since memory kinds will be implemented in Year 2, we defer detailed discussion until next year.

Appendix A

Notes for Implementers

The following are possible implementations of template metaprogramming utilities for UPC++ features.

A.1 `future_element_t` and `future_element_moved_t`

```
template<int I, typename T>
struct future_element; // undefined

template<int I, typename T, typename ...U>
struct future_element<I, future<T, U...>> {
  typedef typename future_element<I-1, future<U...>>::type type;
  typedef typename future_element<I-1, future<U...>>::moved_type
    moved_type;
};

template<typename T, typename ...U>
struct future_element<0, future<T, U...>> {
  typedef T type;
  typedef T&& moved_type;
};

template<int I>
struct future_element<I, future<>> {
  typedef void type;
  typedef void moved_type;
};

```

```
template<int I, typename T>
using future_element_t = typename future_element<I, T>::type;

template<int I, typename T>
using future_element_moved_t =
  typename future_element<I, T>::moved_type;
```

A.2 future<T...>::when_all

Utility types:

```
template<template<typename ...Us> class T, typename A, typename B>
struct concat_type; // undefined

template<template<typename ...Us> class T,
         typename ...As, typename... Bs>
struct concat_type<T, T<As...>, T<Bs...> > {
  typedef T<As..., Bs...> type;
};

template<template<typename ...Us> class T,
         typename A, typename... Bs>
struct concat_element_types {
  typedef typename concat_element_types<T, Bs...>::type rest;
  typedef typename concat_type<T, A, rest>::type type;
};

template<template<typename ...Us> class T, typename A>
struct concat_element_types<T, A> {
  typedef A type;
};

template<template<typename ...Us> class T, typename ...U>
using concat_element_types_t =
  typename concat_element_types<T, U...>::type;
```

Declaration of `future<T...>::when_all`:

```
template<typename ...Futures>
concat_element_types_t<future, Futures...> when_all(Futures ...fs);
```

A.3 to_future

Utility types:

```
template<typename T>
struct future_type {
  typedef future<T> type;
};

template<typename ...T>
struct future_type<future<T...>> {
  typedef future<T...> type;
};

template<>
struct future_type<void> {
  typedef future<> type;
};

template<typename T>
using future_type_t = typename future_type<T>::type;

template<typename ...T>
using future_types_t =
  concat_element_types_t<future, future_type_t<T>...>;
```

Declaration of `to_future`:

```
template<typename ...U>
future_types_t<U...> to_future(U ...futures_or_results);
```

A.4 future_invoke_result_t

C++11-compliant implementation:

```
template<typename Func, typename... ArgTypes>
using future_invoke_result_t =
  future_type_t<typename std::result_of<Func(ArgTypes...)>::type>;
```

C++17-compliant implementation:

```
template<typename Func, typename... ArgTypes>
using future_invoke_result_t =
  future_type_t<std::invoke_result_t<Func, ArgTypes...>>;
```

A.5 promise_invoke_result_t

Utility types:

```
template<typename T>
struct promise_type {
  typedef promise<T> type;
};

template<typename ...T>
struct promise_type<future<T...>> {
  typedef promise<T...> type;
};

template<>
struct promise_type<void> {
  typedef promise<> type;
};

template<typename T>
using promise_type_t = typename promise_type<T>::type;
```

C++11-compliant implementation:

```
template<typename Func, typename... ArgTypes>
using promise_invoke_result_t =
  promise_type_t<typename std::result_of<Func(ArgTypes...)>::type>;
```

C++17-compliant implementation:

```
template<typename Func, typename... ArgTypes>
using promise_invoke_result_t =
  promise_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.