UPC++ Programmer's Guide (v2019.3.0)

Lawrence Berkeley National Laboratory Technical Report (LBNL-2001191)

Contents

Acknowledgments

This work 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.

1 Introduction

UPC++ is a C++11 library that supports Partitioned Global Address Space (PGAS) programming. It is designed for writing efficient, scalable parallel programs on distributed-memory parallel computers. The PGAS model is single program, multiple-data (SPMD) in which each separate thread of execution (referred to as a *process*) has access to private memory as well as a global address space. This global address space is accessible to all processes and is allocated in shared segments that are distributed over the processes (see Figure 1). UPC++ provides various convenient methods for accessing and using this global memory, as will be described later in this guide. In UPC++, all accesses to remote memory are explicit, via a special set of methods. There is no implicit communication. This design decision was made to encourage programmers to be aware of the cost of data movement, which may incur expensive communication. Moreover, all remote-memory access operations are asynchronous by default. Together, these two constraints are intended to enable programmers to write code that performs well at scale.

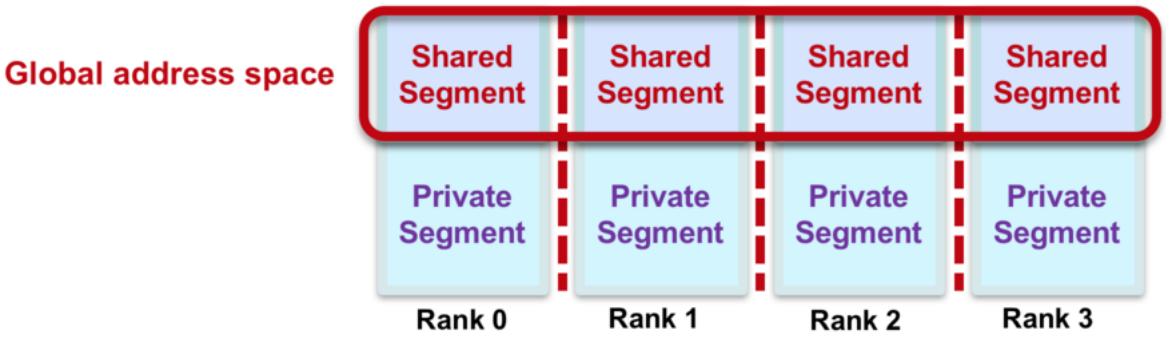

Figure 1: *Figure 1. PGAS Memory Model.*

This guide describes the LBNL implementation of UPC++, which uses GASNet for communication across a wide variety of platforms, ranging from Ethernet-connected laptops to commodity InfiniBand clusters and supercomputers with custom high-performance networks. GASNet is a language-independent, networking middleware layer that provides network-independent, high-performance communication primitives tailored for implementing parallel global address space languages and libraries such as UPC, UPC++, Co-Array Fortran, Legion, Chapel, and many others. For more information about GASNet, visit https://gasnet.lbl.gov.

Although our implementation of UPC++ uses GASNet, in this guide, only the Getting Started with UPC++ section is specific to the implementation. The LBNL implementation of UPC++ adheres to the implementation-independent specification. Both are available at the UPC++ homepage at https://upcxx.lbl.gov/. Please report any problems in the issue tracker, https://upcxx.lbl.gov/issues.

UPC++ has been designed with modern object-oriented concepts in mind. Novices to C++ should avail themselves of good-quality tutorials and documentation to refresh their knowledge of Template Meta programming, the C++ standard library (`std::`), and lambda functions, which are used heavily in UPC++.

2 Getting Started with UPC++

We present a brief description of how to install UPC++ and compile and run UPC++ programs. For more detail, consult the INSTALL.md file that comes with the distribution.

2.1 Installing UPC++

This programming guide assumes that the source code file has been extracted to a directory, `<upcxx-source-path>`. From the top-level of this directory, run the `install` script:

```
./install <upcxx-install-path>
```

This will build the UPC++ library and install it to the `<upcxx-install-path>` directory. We recommend that users choose an installation path which is a non-existent or empty directory path, so that uninstallation is as trivial as `rm -rf <upcxx-install-path>`. Note that the normal installer downloads the GASNet communication library, so an Internet connection is required. An offline installer is also available.

For Mac installations, the Xcode Command Line Tools need to be installed *before* invoking `install`, i.e.:

```
xcode-select --install
```

To build for the compute nodes of a Cray XC, the `CROSS` environment variable needs to be set before the install command is invoked, i.e. `CROSS=cray-aries-slurm`, e.g.:

```
cd <upcxx-source-path>
CROSS=cray-aries-slurm ./install <upcxx-install-path>
```

The installer will use the `cc` and `CC` compiler aliases of the loaded Cray programming environment.

The list of compatible versions of compilers for the various platforms can be found in the README.md that comes with the distribution, under the section "System Requirements". The `install` script checks that the compiler is supported and if not, it terminates with an error message indicating that `CXX` and `CC` need to be set to supported compilers.

2.2 Compiling UPC++ Programs

To compile against UPC++, the `<upcxx-install-path>/bin/upcxx` wrapper can be used. For example, to build the hello world code given previously, execute:

```
<upcxx-install-path>/bin/upcxx -O hello-world.cpp
```

Note that either the `-O` (for optimized) or `-g` (for debug mode) will need to be specified. Alternatively, instead of `-O` or `-g`, the environment variable `UPCXX_CODEMODE={O3,debug}` can be set. If C++14 or C++17 specific features are required, the C++ standard will also need to be specified (either `-std=c++14` or `-std=c++17`). For an example, look at the `Makefile` in the `<upcxx-source-path>/example/prog-guide/` directory. That directory also has code for all of the examples given in this guide. To use the `Makefile`, first set the `UPCXX_INSTALL` shell variable to the install path.

UPC++ also supports multithreading within a process, e.g. using OpenMP. In these cases, to ensure that the application is compiled against a thread-safe UPC++ backend, set the environment variable `UPCXX_THREADMODE=par`. Note that this option is less efficient than the default, `UPCXX_THREADMODE=seq`, which enables the use of a UPC++ backend that is synchronization-free in most of its internals; thus, the parallel thread mode should only be used when multithreading within processes.

In general, the network conduit (a particular implementation of the interconnection network, provided by GASNet, upon which UPC++'s backend is implemented) is automatically set and shouldn't have to be changed. However, it can be explicitly set using the `UPCXX_GASNET_CONDUIT` variable, e.g. to set the conduit to UDP:

```
export UPCXX_GASNET_CONDUIT=udp
```

More details about both installation and compilation can be found in the `INSTALL.md` file in the source code root directory.

2.3 Running UPC++ Programs

To run a parallel UPC++ application, use the `upcxx-run` launcher provided in the installation directory:

```
<upcxx-install-path>/bin/upcxx-run -n <processes> <exe> <args...>
```

The launcher will run the executable and arguments `<exe> <args...>` in a parallel context with `<processes>` number of UPC++ processes. For multiple nodes, specify the node count with `-N <nodes>`.

Upon startup, each UPC++ process creates a fixed-size shared memory heap that will never grow. By default, this heap is 128 MB per process. This heap size can be set at startup by passing a `-shared-heap` parameter to the run script, which takes a suffix of KB, MB or GB; e.g. to reserve 1GB per process, call:

```
<upcxx-install-path>/bin/upcxx-run -shared-heap 1G -n <processes> <exe> <args...>
```

One can also specify the shared heap size as a percentage of physical memory, split evenly between processes sharing the same node, e.g., `-shared-heap=50%`. There are several other options that can be passed to `upcxx-run`. Execute with `-h` to get the following list of options:

```
usage: upcxx-run [-h] [-n NUM] [-N NUM] [-shared-heap HEAPSZ] [-backtrace]
                 [-show] [-info] [-ssh-servers HOSTS] [-localhost] [-v] [-vv]
                 command ...

A portable parallel job launcher for UPC++ programs, v2018.9.0

options:
  -h, --help            show this help message and exit
  -n NUM, -np NUM       Spawn NUM number of UPC++ processes. Required.
  -N NUM                Run on NUM of nodes.
  -shared-heap HEAPSZ   Requests HEAPSZ size of shared memory per process.
                        HEAPSZ must include a unit suffix matching the
                        pattern "[KMGT]B?" or be "[0-100]%" (case-insensitive).
  -backtrace            Enable backtraces. Compile with -g for full debugging
                        information.
  -show                 Testing: do not start the job, just output the command
                        line that would have been executed
  -info                 Display useful information about the executable
  -ssh-servers HOSTS    List of SSH servers, comma separated.
  -localhost            Run UDP-conduit program on local host only
  -v                    Verbose output
  -vv                   Very verbose output

command to execute:
  command               UPC++ executable
  ...                   arguments
```

3 Hello World in UPC++

The following code implements "Hello World" in UPC++:

```
#include <iostream>
#include <upcxx/upcxx.hpp>

// we will assume this is always used in all examples
using namespace std;

int main(int argc, char *argv[])
{
  // setup UPC++ runtime
  upcxx::init();
  cout << "Hello world from process " << upcxx::rank_me()
```

```
      << " out of " << upcxx::rank_n() << " processes" << endl;
  // close down UPC++ runtime
  upcxx::finalize();
  return 0;
}
```

The UPC++ runtime is initialized with a call to `upcxx::init()`, after which there are multiple processes running, each executing the same code. The runtime must be closed down with a call to `upcxx::finalize()`. In this example, the call to `upcxx::rank_me()` gives the index for the running process, and `upcxx::rank_n()` gives the total number of processes. The use of *rank* in the function names refers to the rank within a team, which in this case contains all processes, i.e. team `upcxx::world`. Teams are described in detail in the Teams section.

When this "hello world" example is run on four processes, it will produce output something like the following (there is no expected order across the processes):

```
Hello world from process 0 out of 4 processes
Hello world from process 2 out of 4 processes
Hello world from process 3 out of 4 processes
Hello world from process 1 out of 4 processes
```

4 Global Memory

A UPC++ program can allocate global memory in shared segments, which are accessible by all processes. A global pointer points at storage within the global memory, and is declared as follows:

```
upcxx::global_ptr<int> gptr = upcxx::new_<int>(upcxx::rank_me());
```

The call to `upcxx::new_<int>` allocates a new integer in the calling process's shared segment, and returns a global pointer (`upcxx::global_ptr`) to the allocated memory. This is illustrated in figure 2, which shows that each process has its own private pointer (`gptr`) to an integer in its local shared segment. By contrast, a conventional C++ dynamic allocation (`int *mine = new int`) will be in private local memory. Note that we use the integer type in this paragraph as an example, but any type `T` can be allocated using the `upcxx::new_<T>()` function call.

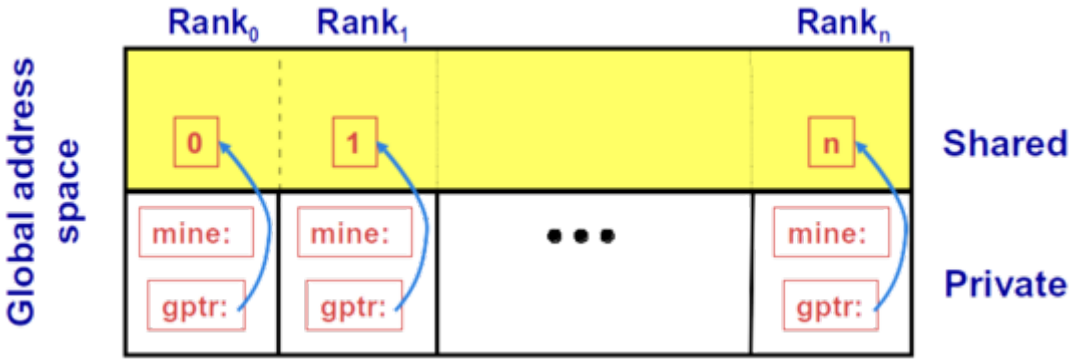

Figure 2: *Figure 2. Global pointers.*

A UPC++ global pointer is fundamentally different from a conventional C++ pointer: it cannot be dereferenced using the `*` operator; it does not support conversions between pointers to base and derived types; and it cannot be constructed by the C++ `std::addressof` operator. However, UPC++ global pointers support pointer arithmetic and may be passed by value.

The `upcxx::new_` function calls the class constructor in addition to allocating memory. Since we are allocating a scalar, we can pass arguments to the constructor. Thus, we don't have to invoke the default constructor. The `upcxx::new_` function is paired with `upcxx::delete_`:

```
upcxx::global_ptr<int> x = upcxx::new_<int>(42);
// work with x
...
upcxx::delete_(x);
```

In addition, UPC++ provides a function, `upcxx::new_array`, for allocating a 1-dimensional array in the global address space. Note that this array is not distributed across processes (as in UPC). Rather, each process allocates its own array object which need not have the same size. The `upcxx::new_array` operation calls the *default* class constructor for the objects being allocated. The destruction operation is `upcxx::delete_array`; for example, to allocate 10 initialized elements:

```
upcxx::global_ptr<int> x_arr = upcxx::new_array<int>(10);
// work with x_arr
...
upcxx::delete_array(x_arr);
```

UPC++ also provides functions for allocating and deallocating uninitialized shared storage without calling constructors and destructors. The `upcxx::allocate` function allocates enough (uninitialized) space for `n` shared objects of type `T` on the current process, with a specified alignment, and `upcxx::deallocate` frees the memory.

4.1 Downcasting global pointers

If the shared memory pointed to by a global pointer has affinity to a local process, UPC++ enables a programmer to use a global pointer as an ordinary C++ pointer, through downcasting, via the `local` method of `upcxx::global_ptr`, for example:

```
upcxx::global_ptr<int> x_arr = upcxx::new_array<int>(10);
assert(x_arr.is_local());   // a precondition of global_ptr<T>.local()
int *local_ptr = x_arr.local();
local_ptr[i] = ... // work with local ptr
```

Using this downcasting feature, we can treat all shared objects allocated by a process as local objects (global references are needed by remote processes only). Storage with local affinity can be accessed more efficiently via an ordinary C++ pointer. Note that the `local()` method is safe only if the global pointer is local. To this end, UPC++ provides the `is_local` method for global pointer to check for locality, which will return true if this global pointer is local and can be downcast. This call can help to avoid catastrophic errors, as it is is incorrect to attempt to downcast a global pointer that is *not* local to a process. It's a good idea to make liberal use of `assert(gptr.is_local())` when downcasting pointers to ensure that the downcast is valid.

5 Using Global Memory with One-sided Communication

We illustrate the use of global shared memory with a simple program that implements a one-dimensional (1D) stencil. Stencil codes are widely used for modeling physical systems, e.g. for solving differential equations. Stencil codes are iterative algorithms that operate on a mesh or partition of data, where the value at each mesh location is computed based on the values at neighboring locations. A trivial example is where at each iteration the value for each mesh location `u[i]` is computed as the average of its neighbors:

```
u[i] = (u[i - 1] + u[i + 1]) / 2;
```

In many applications, these updates are iterated until the computed solution converges to a desired degree of accuracy, `epsilon`.

One way to parallelize the 1D stencil computation is with *red-black* updates. The mesh is divided into two groups: the odd-numbered indexes (red) and the even-numbered indexes (black). When iterating, first all the red indexes are updated, and then all the black. This enables us to compute the values for all of a color in any order with only dependencies on the other color, as shown in the following code sketch:

```
for (int step = 0; step < max_steps; step++) {
  for (int i = 1 + step % 2; i < n - 1; i += 2)
    u[i] = (u[i - 1] + u[i + 1]) / 2;
  if (error(u) <= epsilon) break;
}
```

Note that the red updates are done on even-numbered steps, and the black updates on odd-numbered.

To implement the parallel algorithm in UPC++, we split the solution into *panels*, one per process. Each process can operate independently on its own subset of the mesh, except at the boundaries, where a process will need to access the neighbors' values. In order to do this, we can allocate the panels in the shared segment, using UPC++'s array allocation:

```
upcxx::global_ptr<double> u_gptr = upcxx::new_array<double>(n_local);
```

Here `n_local` is the number of points assigned to the local panel. Assuming that the total number of processes divides `n` evenly, we have `n_local = n / upcxx::rank_n() + 2`. The final `+2` is extra space for ghost cells to store the neighboring values.

Processes in UPC++ are not automatically aware of new allocations in another process's shared segment, so we must ensure that the processes obtain the global pointers that they require. There are several different ways of doing this; we will show how to do it using a convenient construct provided by UPC++, called a *distributed object*.

A distributed object is a single logical object partitioned over a set of processes (a team), where every process has the same global name for the object (i.e. a universal name), but its own local value. Distributed objects are created with the `upcxx::dist_object<T>` type, for example, in our stencil, we can declare our panels to be a distributed object:

```
upcxx::dist_object<upcxx::global_ptr<double>> u_g(upcxx::new_array<double>(n_local));
```

Each process in a given team must call a constructor collectively for `upcxx::dist_object<T>`, with a value of type `T` representing the process's instance value for the object (a global pointer to an array of doubles in the example above). Although the constructor for a distributed object is collective, there is no guarantee that when the constructor returns on a given process it will be complete on any other process. To avoid this hazard, UPC++ provides an interlock to ensure that accesses to a `dist_object` are delayed until the local representative has been constructed.

We can still use the `local` method of `upcxx::global_ptr` to downcast the pointer contained in a distributed object, provided the allocation has affinity to the process doing the downcast. We use this feature to get an ordinary C++ pointer to the panel:

```
double *u = u_g->local();
```

To access the remote value of a distributed object, we use the `upcxx::dist_object::fetch` member function, which, given a process rank argument, will get the `T` value from the sibling distributed object representative on the remote process. For simplicity we will use periodic boundary conditions, so process 0 has process `n-1` as its left neighbor, and process `n-1` has process 0 as its right neighbor. In our algorithm, we need to fetch the global pointers for the left (`uL`) and right (`uR`) ghost cells:

```
int left_nb = (!upcxx::rank_me() ? upcxx::rank_n() - 1 : upcxx::rank_me() - 1);
uL = u_g.fetch(left_nb).wait();
uR = u_g.fetch((upcxx::rank_me() + 1) % upcxx::rank_n()).wait();
```

Because the fetch function is asynchronous, we have to synchronize on completion, using a call to `wait()`. Later, in the Asynchronous Computation section, we will see how to overlap asynchronous operations, that is,

when communication is split-phased. But for now we do not separate the asynchronous initiation from the `wait()`.

Now, when we execute the red-black iterations, we can use the global pointers to the ghost cells to get the remote processes' values. To do this, we invoke a *remote get*:

```
if (!(step % 2)) u[0] = upcxx::rget(uL + block).wait();
else u[n_local - 1] = upcxx::rget(uR + 1).wait();
```

The remote get function is part of the one-sided communication model supported by UPC++. Also supported is a remote put function, `upcxx::rput`. These operations initiate transfer of the `value` object to (put) or from (get) the remote process; no coordination is needed with the remote process (this is why it is *one-sided*). The type `T` transferred must be *TriviallySerializable*, generally by satisfying the C++ *TriviallyCopyable* concept. Like many asynchronous communication operations, `rget` and `rput` default to returning a future object that becomes ready when the transfer is complete (futures are discussed in more detail in the Asynchronous Computation section, and completions in general are described in more detail in the Completions section).

Putting all the components described together, the main function for the red-black solver is:

```
int main(int argc, char **argv)
{
  upcxx::init();
  // initialize parameters - simple test case
  const long N = 1000;
  const long MAX_ITER = N * N * 2;
  const double EPSILON = 0.1;
  const int MAX_VAL = 100;
  const int EXPECTED_VAL = MAX_VAL / 2;
  // get the bounds for the local panel, assuming number of processes divides N evenly
  int block = N / upcxx::rank_n();
  // plus two for ghost cells
  int n_local = block + 2;
  // set up the distributed object
  upcxx::dist_object<upcxx::global_ptr<double>> u_g(upcxx::new_array<double>(n_local));
  // downcast to a regular C++ pointer
  double *u = u_g->local();
  // fill with uniformly distributed random values
  mt19937_64 rgen(upcxx::rank_me() + 1);
  for (int i = 1; i < n_local - 1; i++)
    u[i] = 0.5 + rgen() % MAX_VAL;
  upcxx::global_ptr<double> uL = nullptr, uR = nullptr;
  // fetch the left and right pointers for the ghost cells
  int left_nb = (!upcxx::rank_me() ? upcxx::rank_n() - 1 : upcxx::rank_me() - 1);
  uL = u_g.fetch(left_nb).wait();
  uR = u_g.fetch((upcxx::rank_me() + 1) % upcxx::rank_n()).wait();
  upcxx::barrier();
  // iteratively solve
  for (long stepi = 0; stepi < MAX_ITER; stepi++) {
    // alternate between red and black
    int start = stepi % 2;
    // get the values for the ghost cells
    if (!start) u[0] = upcxx::rget(uL + block).wait();
    else u[n_local - 1] = upcxx::rget(uR + 1).wait();
    // compute updates and error
    for (int i = start + 1; i < n_local - 1; i += 2)
      u[i] = (u[i - 1] + u[i + 1]) / 2.0;
```

```
    // wait until all processes have finished calculations
    upcxx::barrier();
    // periodically check convergence
    if (stepi % 10 == 0) {
      if (check_convergence(u, n_local, EXPECTED_VAL, EPSILON, stepi))
        break;
    }
  }
  upcxx::finalize();
  return 0;
}
```

We have one helper function, `check_convergence`, which determines if the solution has converged. It uses a UPC++ collective, `upcxx::reduce_all`, to enable all the processes to obtain the current maximum error:

```
bool check_convergence(double *u, int n_local, const double EXPECTED_VAL,
                       const double EPSILON, long stepi)
{
  double err = 0;
  for (int i = 1; i < n_local - 1; i++)
    err = max(err, fabs(EXPECTED_VAL - u[i]));
  // upcxx collective to get max error over all processes
  double max_err = upcxx::reduce_all(err, upcxx::op_fast_max).wait();
  // check for convergence
  if (max_err / EXPECTED_VAL <= EPSILON) {
    if (!upcxx::rank_me())
      cout << "Converged at " << stepi <<", err " << max_err << endl;
    return true;
  }
  return false;
}
```

Because the collective function is asynchronous, we synchronize on completion, using `wait()`, to retrieve the result.

6 Remote Procedure Calls

An RPC enables the calling process to invoke a function at a remote process, using parameters sent to the remote process via the RPC. For example, to execute a function `square` on process `r`, we would call:

```
int square(int a, int b) { return a * b; }
upcxx::future<int> fut_result = upcxx::rpc(r, square, a, b);
```

By default, an RPC returns the result in an appropriately typed `upcxx::future`, and we can obtain the value using `wait`, i.e.

```
int result = fut_result.wait();
```

(for more on futures, see the Asynchronous Computation section). The function passed in can be a lambda or another function, but note that the function cannot be in a shared library. The arguments to an RPC must be Serializable, generally either by satisfying the C++ TriviallyCopyable concept (which includes all builtin types and many class types), or be one of the commonly used STL types (see also View-Based Serialization). Support for serialization of more complex types will appear in a future release of UPC++.

We illustrate the use of RPCs with a simple distributed hash table, similar to the C++ unordered map. The hash table is implemented in a header file, and provides insert and find operations. It relies on each process

having a local `std::unordered_map` to store the key-value pairs, and in order for the local unordered maps to be accessible from the RPCs, we wrap them in a distributed object:

```
using dobj_map_t = upcxx::dist_object<std::unordered_map<std::string, std::string> >;
dobj_map_t local_map;
```

This is initialized in the constructor:

```
DistrMap() : local_map({}) {}
```

For the insert operation, the target process is determined by a hash function, `get_target_rank(key)`, which maps the key to one of the processes. The data is inserted using a lambda function, which takes a key and a value as parameters. The insert operation returns the result of the lambda, which in this case is an empty future:

```
upcxx::future<> insert(const std::string &key, const std::string &val) {
  return upcxx::rpc(get_target_rank(key),
                    [](dobj_map_t &lmap, std::string key, std::string val) {
                      lmap->insert({key, val});
                    }, local_map, key, val);
}
```

Note the use of the `dist_object` argument `local_map` to the RPC. As a special case, when a `dist_object` is passed as the argument to an RPC, the RPC activation at the target receives a reference to its local representative of the same `dist_object`.

Special care should be taken by the programmer when using lambda captures for RPC. In particular, *when passing C++ lambdas to the UPC++ RPC operations, reference captures should never be used.* This is because the C++ compiler implements reference captures using raw virtual memory addresses, which are generally not meaningful at the remote process that executes the RPC callback.

The find operation is similar, and is given in the full header example:

```
#include <map>
#include <upcxx/upcxx.hpp>

class DistrMap
{
private:
  // store the local unordered map in a distributed object to access from RPCs
  using dobj_map_t = upcxx::dist_object<std::unordered_map<std::string, std::string> >;
  dobj_map_t local_map;
  // map the key to a target process
  int get_target_rank(const std::string &key) {
    return std::hash<std::string>{}(key) % upcxx::rank_n();
  }
public:
  // initialize the local map
  DistrMap() : local_map({}) {}
  // insert a key-value pair into the hash table
  upcxx::future<> insert(const std::string &key, const std::string &val) {
    // the RPC returns an empty upcxx::future by default
    return upcxx::rpc(get_target_rank(key),
                      // lambda to insert the key-value pair
                      [](dobj_map_t &lmap, std::string key, std::string val) {
                        // insert into the local map at the target
                        lmap->insert({key, val});
                      }, local_map, key, val);
```

```
  }
  // find a key and return associated value in a future
  upcxx::future<std::string> find(const std::string &key) {
    return upcxx::rpc(get_target_rank(key),
                      // lambda to find the key in the local map
                      [](dobj_map_t &lmap, std::string key) -> std::string {
                        auto elem = lmap->find(key);
                        // no key found
                        if (elem == lmap->end()) return std::string();
                        // the key was found, return the value
                        return elem->second;
                      }, local_map, key);
  }
};
```

A test example of using the distributed hash table is shown below. Each process generates a set of `N` key-value pairs, guaranteed to be unique, inserts them all into the hash table, and then retrieves the set of keys for the next process over, asserting that each one was found and correct. The insertion phase is followed by a barrier, to ensure all insertions have completed:

```
#include <iostream>
#include "dmap.hpp"

using namespace std;

int main(int argc, char *argv[])
{
  upcxx::init();
  const long N = 10000;
  DistrMap dmap;
  // insert set of unique key, value pairs into hash map, wait for completion
  for (long i = 0; i < N; i++) {
    string key = to_string(upcxx::rank_me()) + ":" + to_string(i);
    string val = key;
    dmap.insert(key, val).wait();
  }
  // barrier to ensure all insertions have completed
  upcxx::barrier();
  // now try to fetch keys inserted by neighbor
  for (long i = 0; i < N; i++) {
    string key = to_string((upcxx::rank_me() + 1) % upcxx::rank_n()) + ":" + to_string(i);
    string val = dmap.find(key).wait();
    // check that value is correct
    assert(val == key);
  }
  upcxx::barrier(); // wait for finds to complete globally
  if (!upcxx::rank_me()) cout << "SUCCESS" << endl;
  upcxx::finalize();
  return 0;
}
```

In the previous example, the `rpc` operations are synchronous because of the call to the `wait` method. To achieve maximum performance, UPC++ programs should always take advantage of asynchrony when possible.

7 Asynchronous Computation

Most communication operations in UPC++ are asynchronous. So, in our red-black solver example, when we made the call to `upcxx::rget`, we explicitly waited for it to complete using `wait()`. However, in split-phase algorithms, we can perform the wait at a later point, allowing us to overlap computation and communication.

The return type for `upcxx::rget` is dependent on the UPC++ *completion object* passed to the UPC++ call. The default completion is a UPC++ *future*, which holds a value (or tuple of values) and a state (ready or not ready). We will use this default completion here, and will return to the subject in detail in the Completions section. When the `rget` completes, the future becomes ready and can be used to access the results of the operation. The call to `wait()` can be replaced by the following equivalent polling loop, which exits when communication has completed:

```
upcxx::future<double> fut = upcxx::rget(uR + 1);
while (!fut.ready()) upcxx::progress();
u[n_local - 1] = fut.result();
```

First, we get the future object, and then we loop on it until it becomes ready. This loop must include a call to the `upcxx::progress` function, which progresses the library and transitions futures to a ready state when their corresponding operation completes (see the Progress section for more details on progress). This common paradigm is embodied in the `wait()` method of `upcxx::future`.

Using futures, the process waiting for a result can do computation while waiting, effectively overlapping computation and communication. For example, in our distributed hash table, we can do the generation of the key-value pair asynchronously as follows:

```
// initialize key and value for first insertion
string key = to_string(upcxx::rank_me()) + ":" + to_string(0);
string val = key;
for (long i = 0; i < N; i++) {
  upcxx::future<> fut = dmap.insert(key, val);
  // compute new key while waiting for RPC to complete
  if (i < N - 1) {
    key = to_string(upcxx::rank_me()) + ":" + to_string(i + 1);
    val = key;
  }
  // wait for operation to complete before next insert
  fut.wait();
}
```

UPC++ also provides *callbacks* or *completion handlers* that can be attached to futures. These are functions that are executed when the future is ready (this is sometimes also referred to as "future chaining"). In our distributed hash table example, we want to check the value once the `find` operation returns. To do this asynchronously, we can attach a callback to the future using the `.then` method of `upcxx::future`. The callback is executed on the initiating process when the `find` completes, and is passed the result of the future as a parameter. In this example, the callback is a lambda, which checks the returned value:

```
for (long i = 0; i < N; i++) {
  string key = to_string((upcxx::rank_me() + 1) % upcxx::rank_n()) + ":" + to_string(i);
  // attach callback, which itself returns a future
  upcxx::future<> fut = dmap.find(key).then(
    // lambda to check the return value
    [key](string val) {
      assert(val == key);
    });
  // wait for future and its callback to complete
```

```
  fut.wait();
}
```

An important feature of UPC++ is that there are no implicit ordering guarantees with respect to asynchronous operations. In particular, there is no guarantee that operations will complete in the order they were initiated. This allows for more efficient implementations, but the programmer must not assume any ordering that is not enforced by explicit synchronization.

7.1 Conjoining Futures

When many asynchronous operations are launched, it can be tricky to keep track of all the futures, and wait on all of them for completion. In the previous example of asynchronous finds, we might have preferred to issue all the finds asynchronously and wait on all of them to complete at the end. UPC++ provides an elegant solution to do this, allowing futures to be *conjoined* (i.e. aggregated), so that the results of one future are dependent on others, and we need only wait for completion explicitly on one future. The example below illustrates how the hash table lookups can be performed asynchronously using this technique:

```
// the start of the conjoined future
upcxx::future<> fut_all = upcxx::make_future();
for (long i = 0; i < N; i++) {
  string key = to_string((upcxx::rank_me() + 1) % upcxx::rank_n()) + ":" + to_string(i);
  // attach callback, which itself returns a future
  upcxx::future<> fut = dmap.find(key).then(
    // lambda to check the return value
    [key](string val) {
      assert(val == key);
    });
  // conjoin the futures
  fut_all = upcxx::when_all(fut_all, fut);
  // periodically call progress to allow incoming RPCs to be processed
  if (i % 10 == 0) upcxx::progress();
}
// wait for all the conjoined futures to complete
fut_all.wait();
```

The future conjoining begins by invoking `upcxx::make_future` to construct a trivially ready future `fut_all`. We then loop through each iteration, calling `DistrMap::find` asynchronously, and obtaining the future `fut` returned by the `.then` callback attached to the `find` operation. This future is passed to the `upcxx::when_all` function, in combination with the previous future, `fut_all`. The `upcxx::when_all` constructs a new future representing readiness of all its arguments (in this case `fut` and `fut_all`), and returns a future with a concatenated results tuple of the arguments. By setting `fut_all` to the future returned by `when_all`, we can extend the conjoined futures. Once all the processes are linked into the conjoined futures, we simply wait on the final future, i.e. the `fut_all.wait()` call. Note that within the loop, we have a periodic call to `upcxx::progress`, which gives the UPC++ runtime an opportunity to process incoming RPCs and run completion callbacks (progress is described in more detail in the Progress section).

It is also possible to keep track of futures with an array or some other container, such as a vector. However, conjoining futures has several advantages:

- Only a single future handle is needed to manage an entire logical operation and associated storage. This simplifies composition by enabling an encapsulated communication initiation function to return a single future to the caller rather than, for instance, an array of futures.
- Conjoined futures minimize the number of synchronous calls to `.wait()` needed to complete a group of operations. Without conjoined futures, the code must loop through multiple `.wait` calls, potentially adding overhead.

- Conjoined futures do not leave intermediate futures explicitly instantiated in the application-level memory, thus potentially allowing the storage holding them to be reclaimed as soon as the futures are ready.

8 Quiescence

Quiescence is a state in which no process is performing computation that will result in communication injection, and no communication operations are currently in-flight in the network or queues on any process. Quiescence is of particular importance for applications using anonymous asynchronous operations on which no synchronization is possible on the sender's side. For example, quiescence may need to be achieved before destructing resources and/or exiting a UPC++ computational phase.

To illustrate a simple approach to quiescence, we use the distributed hash table again. In this case, we use a version of RPC that does not return a future:

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

The "ff" stands for "fire-and-forget". From a performance standpoint, it has the advantage that is does not *send a response message* to satisfy the `future` back to the process which has issued the RPC. However, because no acknowledgment is returned to the initiator, the caller does not get a future to indicate when the `upcxx::rpc_ff` invocation has completed execution at the target. We can modify the distributed hash table insert operation to use `upcxx::rpc_ff` as follows:

```
// insert a key-value pair into the hash table
void insert(const std::string &key, const std::string &val) {
  // this RPC does not return anything
  upcxx::rpc_ff(get_target_rank(key),
                // lambda to insert the key-value pair
                [](dobj_map_t &lmap, std::string key, std::string val) {
                  // insert into the local map at the target
                  lmap->insert({key, val});
                }, local_map, key, val);
}
```

The only way to determine completion is to use additional code; for example, we could change the distributed hash table insert loop as follows:

```
// distributed object to keep track of number of inserts expected at this process
upcxx::dist_object<long> n_inserts = 0;
// keep track of how many inserts have been made to each target process
std::unique_ptr<long[]> inserts_per_rank(new long[upcxx::rank_n()]());
// insert all key-value pairs into the hash map
for (long i = 0; i < N; i++) {
  string key = to_string(upcxx::rank_me()) + ":" + to_string(i);
  string val = key;
  // insert has no return because it uses rpc_ff
  dmap.insert(key, val);
  // increment the count for the target process
  inserts_per_rank[dmap.get_target_rank(key)]++;
  // periodically call progress to allow incoming RPCs to be processed
  if (i % 10 == 0) upcxx::progress();
}
// update all remote processes with the expected count
for (long i = 0; i < upcxx::rank_n(); i++) {
```

```
    if (inserts_per_rank[i]) {
      // use rpc to update the remote process's expected count of inserts
      upcxx::rpc(i,
                 [](upcxx::dist_object<long> &e_inserts, long count) {
                   *e_inserts += count;
                 }, n_inserts, inserts_per_rank[i]).wait();
    }
  }
  // wait until all threads have updated insert counts
  upcxx::barrier();
  long expected_inserts = *n_inserts;
  // wait until we have received all the expected updates, spinning on progress
  while (dmap.local_size() < expected_inserts) upcxx::progress();
```

To track completion each process must know how many inserts to expect. Thus each process keeps track of how many inserts it makes to every other process in the local variable `inserts_per_rank`. Once a process has finished all its inserts, it uses an RPC to update the value of a distributed object, `n_inserts` for all the destination processes. After updating this counter, all processes wait on a barrier, which ensures that all processes have *dispatched* their inserts. However, the barrier does not guarantee that all the inserts in flight have completed at their destinations. To ensure this is the case, every process must spin on a comparison of the expected counts to the size of the local unordered map, given by the method `DistrMap::local_size()`, which is defined simply as:

```
int local_size() { return local_map->size(); }
```

Within the while loop, `upcxx::progress` is called each time to ensure that the UPC++ runtime engine executes any outstanding RPCs that have arrived at the process (progress is described in more detail in the Progress section).

This is one way to achieve quiescence; there are many alternatives. When the number of messages to be received is known beforehand or can be efficiently tracked, it is possible to implement simple mechanisms such as the one used in our distributed hash table example. There are also more powerful (and more scalable) quiescence algorithms, such as the *counting algorithm*, that do not require knowledge of the number of messages/RPCs beforehand. Advanced users requiring this capability should consult the appropriate literature.

9 Atomics

UPC++ provides atomic operations on shared objects. These operations are handled differently from C++ atomics and rely on the notion of an *atomic domain,* a concept inherited from UPC. In UPC++, all atomic operations are associated with an atomic domain, an object that encapsulates a supported type and set of operations. Currently, the allowed types are `std::int32_t`, `std::uint32_t`, `std::int64_t`, and `std::uint64_t`. The full list of operations can be found in the UPC++ specification.

Each atomic domain is a collective object comprised of instances of the `atomic_domain` class, and the operations are defined as methods on that class. The use of atomic domains permits selection (at construction) of the most efficient available implementation which can provide correct results for the given set of operations on the given data type. This implementation may be hardware-dependent and vary for different platforms. To get the best possible performance from atomics, the user should be aware of which atomics are supported in hardware on their platform, and set up the domains accordingly.

Similar to a mutex, an atomic domain exists independently of the data it applies to. User code is responsible for ensuring that data accessed via a given atomic domain is only accessed via that domain, never via a different domain or without use of a domain. Users may create as many domains as needed to describe their uses of atomic operations, so long as there is at most one domain per atomic datum. If distinct data of the

same type are accessed using differing sets of operations, then creation of distinct domains for each operation set is recommended to achieve the best performance on each set.

We illustrate atomics by showing how they can be used to solve the quiescence counting problem for the distributed hash table, discussed in the previous section. The code is as follows:

```
  // keep track of how many inserts have been made to each target process
  std::unique_ptr<int64_t[]> inserts_per_rank(new int64_t[upcxx::rank_n()]());
  // insert all key-value pairs into the hash map
  for (long i = 0; i < N; i++) {
    string key = to_string(upcxx::rank_me()) + ":" + to_string(i);
    string val = key;
    dmap.insert(key, val);
    inserts_per_rank[dmap.get_target_rank(key)]++;
     // periodically call progress to allow incoming RPCs to be processed
    if (i % 10 == 0) upcxx::progress();
 }
  // setup atomic domain with only the operations needed
  upcxx::atomic_domain<int64_t> ad({upcxx::atomic_op::load, upcxx::atomic_op::add});
  // distributed object to keep track of number of inserts expected at every process
  upcxx::dist_object<upcxx::global_ptr<int64_t> > n_inserts(upcxx::new_<int64_t>(0));
  // get pointers for all other processes and use atomics to update remote counters
  for (long i = 0; i < upcxx::rank_n(); i++) {
    if (inserts_per_rank[i]) {
      upcxx::global_ptr<int64_t> remote_n_inserts = n_inserts.fetch(i).wait();
      // use atomics to increment the remote process's expected count of inserts
      ad.add(remote_n_inserts, inserts_per_rank[i], memory_order_relaxed).wait();
    }
  }
  upcxx::barrier();
  // Note: once a memory location is accessed with atomics, it should only be
  // subsequently accessed using atomics to prevent unexpected results
  int64_t expected_inserts = ad.load(*n_inserts, memory_order_relaxed).wait();
  // wait until we have received all the expected updates, spinning on progress
  while (dmap.local_size() < expected_inserts) upcxx::progress();
```

We modify the code from the example in the Quiescence section so that the `n_inserts` is actually a pointer to `int64_t`, rather than a `long` itself. We do this because atomic operations require global pointers, and we explicitly use `int64_t` because `long` is not guaranteed to be supported for atomic operations on all platforms. Then we declare an atomic domain, which includes only the two operations we will actually use:

```
  upcxx::atomic_domain<long> ad_long({upcxx::atomic_op::load,
                                      upcxx::atomic_op::add});
```

Each process maintains an array, `inserts_per_rank`, of the expected counts for all other processes. Once it has finished all its inserts, it loops over all the remote processes, and for each one it first obtains the remote global pointer using `fetch`, and then atomically updates the target process's counter. Finally, each process spins, waiting for the size of the local unordered map to match its expected number of inserts.

Like all atomic operations (and indeed, nearly all UPC++ communication operations), the atomic load and add operations are asynchronous. The load operation returns a completion object, which defaults to a future. In the example in this section, we simply waited for each atomic operation to complete, but we could easily use future conjoining or promises to track completion of the atomic operations.

Note that an atomic domain must be explicitly destroyed via a collective call to the `atomic_domain::destroy()` method, before the allocated `atomic_domain` object goes out of scope or is deleted. (This action meets the precondition of the object's destructor, that the object has been destroyed) If the `atomic_domain`

object hasn't been destroyed appropriately, the program will crash with an error message that `upcxx::atomic_domain::destroy()` must be called collectively before the destructor, which has been invoked implicitly.

10 Completions

In the previous examples in this guide, we have relied on futures to inform us about the completion of asynchronous operations. However, UPC++ provides several additional mechanisms for determining completion, including *promises*, *remote procedure calls* (RPCs) and *local procedure calls* (LPCs). Further on in this section, we give an example of completions using promises.

A completion object is constructed by a call to a static member function of one of the completion classes: `upcxx::source_cx`, `upcxx::remote_cx` or `upcxx::operation_cx`. These classes correspond to the different stages of completion of an asynchronous operation: source and remote completions indicate that the resources needed for the operation are no longer in use by UPC++ at the source process or remote process, respectively, whereas operation completion indicates that the operation is complete from the perspective of the initiator. As we have seen in the previous examples in this guide, most operations default to notification of operation completion using a future, e.g.:

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

As shown above, the completion object is constructed as a completion stage combined with a type of completion. There are restrictions on which completion types can be associated with which stages: futures, promises and LPCs are only valid for source and operation completions, whereas RPCs are only valid for remote completions.

It is possible to request multiple completion notifications from one operation using the pipe (`|`) operator to combine completion objects. For example, future completion can be combined with RPC completion as follows:

```
auto cxs = (upcxx::remote_cx::as_rpc(some_func) | upcxx::operation_cx::as_future());
```

The completion argument passed to an asynchronous communication initiation function influences the polymorphic return type of that function (abbreviated as `RType` in prototypes). In particular, if one future completion is requested (as with the default completion), then the return type `RType` is an appropriately typed future (as we've seen in prior examples). If a completion argument is passed that requests no future-based completions, then the return type `RType` is `void`. If more than one future completion is requested (for example, source and operation completions) then the return type `RType` is a `std::tuple` of the requested futures:

```
upcxx::future<> fut_src, fut_op;
std::tie(fut_src, fut_op) = upcxx::rput(p_src, gptr_dst, 1,
               upcxx::source_cx::as_future() | upcxx::operation_cx::as_future());
fut_src.wait();
// ... source memory now safe to overwrite
fut_op.wait(); // wait for the rput operation to be fully complete
```

In the following example, we show how promises can be used to track completion of the distributed hash table inserts. Each non-blocking 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.

First, we modify the `insert` operation to use a promise instead of return a future:

```
// insert a key, value pair into the hash table, track completion with promises
void insert(const std::string &key, const std::string &val, upcxx::promise<> &prom) {
  upcxx::rpc(get_target_rank(key),
             // completion is a promise
             upcxx::operation_cx::as_promise(prom),
             // lambda to insert the key, value pair
             [](dobj_map_t &lmap, std::string key, std::string val) {
               // insert into the local map at the target
               lmap->insert({key, val});
             }, local_map, key, val);
}
```

Now we modify the insert loop to use promises instead of futures for tracking completion:

```
// create an empty promise, to be used for tracking operations
upcxx::promise<> prom;
// insert all key, value pairs into the hash map
for (long i = 0; i < N; i++) {
  string key = to_string(upcxx::rank_me()) + ":" + to_string(i);
  string val = key;
  // pass the promise to the dmap insert operation
  dmap.insert(key, val, prom);
  // periodically call progress to allow incoming RPCs to be processed
  if (i % 10 == 0) upcxx::progress();
}
// finalize the promise
upcxx::future<> fut = prom.finalize();
// wait for the operations to complete
fut.wait();
```

In our distributed hash table example, we create an empty promise, `prom`, which has a dependency count of one. Then we register each `insert` operation in turn on the promise, incrementing the dependency count each time, representing the unfulfilled results of the RPC operation used in the insert. Finally, when registration is complete, the original dependency is fulfilled to signal the end of the registration, with the `prom.finalize` call. This call returns the future associated with the promise, so we can now wait on that future for all the operations to complete.

11 Progress

Progress is a key notion of UPC++ which programmers must be aware of. The UPC++ framework does not spawn any hidden OS threads to advance its internal state or track outstanding asynchronous communication. The rationale is this keeps the performance characteristics of the UPC++ runtime as lightweight and configurable as possible, and simplifies synchronization. Without its own threads, UPC++ is reliant on each application process to periodically grant it access to a thread so that it may make "progress" on its internals. Progress is divided into two levels: *internal progress* and *user-level progress*. With internal progress, UPC++ may advance its internal state, but no notifications will be delivered to the application. Thus the application cannot easily track or be influenced by this level of progress. With user-level progress, UPC++ may advance its internal state as well as signal completion of user-initiated operations. This could include many things: readying futures, running callbacks dependent on those futures, or invoking inbound RPCs.

The `upcxx::progress` function, as already used in our examples, is the primary means by which the application performs the crucial task of temporarily granting UPC++ a thread:

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

A call to `upcxx::progress()` with its default arguments will invoke user-level progress. These calls make for the idiomatic points in the program where the user should expect locally registered callbacks and remotely injected RPCs to execute. There are also other UPC++ functions which invoke user-progress, notably including `upcxx::barrier` and `upcxx::future::wait`. For the programmer, understanding where these functions are called is crucial, since any invocation of user-level progress may execute RPCs or callbacks, and the application code run by those will typically expect certain invariants of the process's local state to be in place.

Many UPC++ operations have a mechanism to signal completion to the application. However, for performance-oriented applications, UPC++ provides an additional asynchronous operation status indicator called `progress_required`. This status indicates that further advancements of the current process or thread's internal-level progress are necessary so that completion of outstanding operations on remote entities (e.g. notification of delivery) can be reached. Once the `progress_required` state has been left, UPC++ guarantees that remote processes will see their side of the completions without any further progress by the current process. The programmer can query UPC++ to determine whether all operations initiated by this process have reached a state at which they no longer require progress using the following function:

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

UPC++ provides a function called `upcxx::discharge()` which polls on `upcxx::progress_required()` and asks for internal progress until progress is not required anymore. `upcxx::discharge()` is equivalent to the following code:

```
while(upcxx::progress_required())
    upcxx::progress(upcxx::progress_level::internal);
```

Any application entering a long lapse of inattentiveness (e.g. to perform expensive computations) is highly encouraged to call `upcxx::discharge()` first.

12 Personas

As mentioned earlier, UPC++ does not spawn background threads for progressing asynchronous operations, but rather leaves control of when such progress is permissible to the user. To help the user in managing the coordination of internal state and threads, UPC++ introduces the concept of *personas*. An object of type `upcxx::persona` represents a collection of UPC++'s internal state. Each persona may be *active* with at most one thread at any time. The active personas of a thread are organized in a stack, with the top persona denoted the *current* persona of that thread. When a thread enters progress, UPC++ will advance the progress of all personas it holds active. When a thread initiates an asynchronous operation, it is registered in the internal state managed by the current (top) persona.

When a thread is created, it is assigned a *default persona*. This persona is always at the bottom of the persona stack, and *cannot be pushed onto the persona stack of another thread*. The default persona can be retrieved using:

```
upcxx::persona& upcxx::default_persona();
```

Additional personas can also be created by the application and pushed onto a thread's persona stack, as described below.

For any UPC++ operation issued by the current persona, the completion notification (e.g. future readying) will be sent to that same persona. This is still the case even if that `upcxx::persona` object has been transferred to a different thread by the time the asynchronous operation completes. The key takeaway here is that a `upcxx::persona` can be used by one thread to issue operations, then passed to another thread (together with the futures corresponding to these operations). That second thread will be then be notified of the completion of these operations via their respective futures. This can be used, for instance, to build a

progress thread — a thread dedicated to progressing asynchronous operations. Another possibility is to use completion objects (see Completions) to execute a callback (more precisely a *local procedure call*, or LPC) on another persona when the operation is complete. We recommend using this second option as `upcxx::future` and `upcxx::promise` objects are *not thread-safe* and thus can only be safely referenced by a thread holding the persona used to create these objects.

UPC++ provides a `upcxx::persona_scope` class for modifying the current thread's active stack of personas. The application is responsible for ensuring that a given persona is only ever on one thread's active stack at any given time. Pushing and popping personas from the stack (hence changing the current persona) is accomplished via RAII semantics using the `upcxx::persona_scope` constructor/destructor.
Here are the constructors:

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

template<typename Lock>
upcxx::persona_scope(Lock &lock, upcxx::persona &p);
```

The `upcxx::persona_scope` constructor takes the `upcxx::persona` that needs to be pushed. Only one thread is permitted to use a given persona at a time. To assist in maintaining this invariant, the `upcxx::persona_scope` constructor accepts an optional thread locking mechanism to acquire during construction and release during destruction (the `Lock` template argument can be any type of lock, such as C++ `std::mutex`).

The following example introduces several of these concepts, using two threads to exchange an integer containing a rank id fetched from a neighbor process (defined as `(upcxx::rank_me() + 1)%upcxx::rank_n()`). A `upcxx::persona` object `progress_persona` is first created. This persona object is used by thread called `submit_thread` to construct a completion object which will execute an LPC on the `progress_persona` as the completion notification for the subsequently issued `rget` operation (the value fetched by the rget is passed as an argument to the LPC callback that is eventually queued for execution by `progress_persona`). The `submit_thread` thread then waits and makes progress until the atomic variable `thread_barrier` is set to 1. Meanwhile, another thread called `progress_thread` pushes `progress_persona` onto its persona stack by constructing an appropriate `upcxx::persona_scope` (in this case we know by construction that only one thread will push this persona, so we can safely use the lock-free constructor). This thread then calls `upcxx::progress()` repeatedly until the `done` boolean is set to true within the LPC attached to the completion object.

```
int main () {
  upcxx::init();
  // create a landing zone, and share it through a dist_object
  // allocate and initialize with local rank
  upcxx::dist_object<upcxx::global_ptr<int>> dptrs(upcxx::new_<int>(upcxx::rank_me()));
  upcxx::global_ptr<int> my_ptr = *dptrs;
  // fetch my neighbor's pointer to its landing zone
  upcxx::intrank_t neigh_rank = (upcxx::rank_me() + 1)%upcxx::rank_n();
  upcxx::global_ptr<int> neigh_ptr = dptrs.fetch(neigh_rank).wait();
  // declare an agreed upon persona for the progress thread
  upcxx::persona progress_persona;
  atomic<int> thread_barrier(0);
  bool done = false;
  // create the progress thread
  thread progress_thread( [&]() {
        // push progress_persona onto this thread's persona stack
        upcxx::persona_scope scope(progress_persona);
        // progress thread drains progress until work is done
        while (!done)
          upcxx::progress();
```

```
        cout<<"Progress thread on process "<<upcxx::rank_me()<<" is done"<<endl;
        //unlock the other threads
        thread_barrier += 1;
      });
  // create another thread to issue the rputs
  thread submit_thread( [&]() {
      // create a completion object to execute a LPC on the progress_thread
      // which verifies that the value we got was the rank of our neighbor
      auto cx = upcxx::operation_cx::as_lpc( progress_persona, [&done, neigh_rank](int got) {
              assert(got == neigh_rank);
              //signal that work is complete
              done = true;
            });
      // use this completion object on the rget
      upcxx::rget(neigh_ptr, cx);
      // block here until the progress thread has executed all LPCs
      while(thread_barrier.load(memory_order_acquire) != 1){
        sched_yield();
        upcxx::progress();
      }
    });
  // wait until all threads finish their work
  submit_thread.join();
  progress_thread.join();
  // wait until all processes are done
  upcxx::barrier();
  if ( upcxx::rank_me()==0 )
    cout<<"SUCCESS"<<endl;
  // delete my landing zone
  upcxx::delete_(my_ptr);
  upcxx::finalize();
}
```

The thread initializing UPC++ via a call to `upcxx::init()` (this is usually done in the main function) is also assigned the *master persona* in addition to its *default persona*. The master persona is special in that it is the only persona in each process which can execute RPCs destined for this process and collective operations. A thread can access this persona object by calling:

```
upcxx::persona& upcxx::master_persona();
```

This persona can also be transferred between threads using `upcxx::persona_scope` objects. However, due to its special nature, the thread which was initially assigned the master persona must first release it before other threads can push it. This is done using the `upcxx::liberate_master_persona()` function. This is of particular importance if one desires to implement a *progress thread* which will execute RPCs issued by remote processes.

As an example, we show how to implement such a progress thread in the distributed hash table example. We first modify the DistrMap class so that a completion object is used to track the completion of the RPC issued by the find function. When the RPC completes, a function `func` provided by the caller is executed as an LPC on the `persona` provided by the caller as well:

```
  // find a key and return associated value in a future
  template <typename Func>
  void find(const std::string &key, upcxx::persona & persona, Func func) {
    // the value returned by the RPC is passed as an argument to the LPC
    // used in the completion object
```

```
  auto cx = upcxx::source_cx::as_buffered() | upcxx::operation_cx::as_lpc(persona,func);
  upcxx::rpc(get_target_rank(key),cx,
      // lambda to find the key in the local map
      [](dobj_map_t &lmap, std::string key) -> std::string {
        auto elem = lmap->find(key);
        // no key found
        if (elem == lmap->end()) return std::string();
        // the key was found, return the value
        return elem->second;
      },local_map,key);
}
```

Let's now review how this can be used to implement a progress thread. A thread `progress_thread` is created to execute incoming RPCs issued to this process, as well as operations submitted to a `progress_persona` (in this example, this thread will execute `N` LPCs). The master persona is therefore first released before the creation of the `progress_thread`. Both the master persona and `progress_persona` are pushed onto `progress_thread`'s persona stack by constructing two `upcxx::persona_scope` objects. The progress thread then calls `upcxx::progress` until `lpc_count` reaches 0, and finally sets the atomic variable `thread_barrier` to signal the other threads that all operations have been completed. Concurrently, ten threads are created to perform a total of `N` find operations, using the `progress_persona` to handle the completion of these operations. This is done by executing an LPC on the `progress_persona` which verifies that the received value corresponds to what was expected and decrement the `lpc_count` counter by one. These threads then call `upcxx::progress()` until the progress thread has handled all operations (See Quiescence). It is important to note that the *master persona* needs to be transferred back to the main thread using a `upcxx::persona_scope` in order to call collectives like `upcxx::barrier()` and `upcxx::finalize()`. Finally, note that in this example we use C++11 threads, but the example would work with other threading mechanisms, such as OpenMP or pthreads.

```
// try to fetch keys inserted by neighbor
// note that in this example, keys and values are assumed to be the same
const int num_threads = 10;
thread * threads[num_threads];
// declare an agreed upon persona for the progress thread
upcxx::persona progress_persona;
atomic<int> thread_barrier(0);
int lpc_count = N;
// liberate the master persona to allow the progress thread to use it
upcxx::liberate_master_persona();
// create a thread to execute the assertions while lpc_count is greater than 0
thread progress_thread( [&]() {
    // push the master persona onto this thread's stack
    upcxx::persona_scope scope(upcxx::master_persona());
    // push the progress_persona as well
    upcxx::persona_scope progress_scope(progress_persona);
    // wait until all assertions in LPCs are complete
    while(lpc_count > 0) {
      sched_yield();
      upcxx::progress();
    }
    cout<<"Progress thread on process "<<upcxx::rank_me()<<" is done"<<endl;
    // unlock the other threads
    thread_barrier += 1;
    });
// launch multiple threads to perform find operations
for (int tid=0; tid<num_threads; tid++) {
```

```
  threads[tid] = new thread( [&,tid] () {
      // split the work across threads
      long num_asserts = N / num_threads;
      long i_beg = tid * num_asserts;
      long i_end = tid==num_threads-1?N:(tid+1)*num_asserts;
      for (long i = i_beg; i < i_end; i++) {
        string key = to_string((upcxx::rank_me() + 1) % upcxx::rank_n()) + ":" + to_string(i);
        // attach callback, which itself runs a LPC on progress_persona on completion
        dmap.find(key, progress_persona,
            [key,&lpc_count](string val) {
              assert(val == key);
              lpc_count--;
            });
      }
      // block here until the progress thread has executed all RPCs and LPCs
      while(thread_barrier.load(memory_order_acquire) != 1){
        sched_yield();
        upcxx::progress();
      }
      });
}

// wait until all threads are done
progress_thread.join();
for (int tid=0; tid<num_threads; tid++) {
  threads[tid]->join();
  delete threads[tid];
}
{
  // push the master persona onto the initial thread's persona stack
  // before calling barrier and finalize
  upcxx::persona_scope scope(upcxx::master_persona());
  // wait until all processes are done
  upcxx::barrier();
  if (upcxx::rank_me() == 0 )
    cout<<"SUCCESS"<<endl;
  upcxx::finalize();
}
```

13 Teams

A UPC++ team is an ordered set of processes and is represented by a `upcxx::team` object. For readers familiar with MPI, teams are similar to `MPI_Communicators` and `MPI_Groups`. The default team for most operations is `upcxx::world()` which includes all processes.

Creating a team is a *collective operation* and may be expensive. It is therefore best to do it in the set-up phase of a calculation. New `upcxx::team` objects can be created by splitting another team using the `upcxx::team::split()` member function. This is demonstrated in the following example, which creates teams consisting of the processes having odd and even ranks in `upcxx:world()` by using `(upcxx::rank_me() % 2)` as the color argument to `split()`. It is worth noting that the key argument is used to *sort* the members of the newly created teams, and need not be precisely the new rank as in this example.

```
upcxx::team & world_team = upcxx::world();
```

```
int color = upcxx::rank_me() % 2;
int key = upcxx::rank_me() / 2;
upcxx::team new_team = world_team.split(color, key);
```

A team object has several member functions. The local rank of the calling process within a team is given by the `rank_me()` function, while `rank_n()` returns the number of processes in the team. The global rank (in the `world()` team) of any team member can be retrieved using the `[]` operator.

The `upcxx::team::from_world()` function converts a global rank from the `world()` team into a local rank within a given team. This function takes either one or two arguments:

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

In the first case, the process with rank `world_index` in `world()` *MUST* be part of the team, while in the second overload, the `otherwise` value will be returned if that process is not part of the team. We can therefore modify our example to do the following (for simplicity, we assume an even number of processes):

```
upcxx::team & world_team = upcxx::world();
int color = upcxx::rank_me() % 2;
int key = upcxx::rank_me() / 2;
upcxx::team new_team = world_team.split(color, key);

upcxx::intrank_t local_rank = new_team.rank_me();
upcxx::intrank_t local_count = new_team.rank_n();

upcxx::intrank_t world_rank = new_team[(local_rank+1)%local_count];
upcxx::intrank_t expected_world_rank = (upcxx::rank_me() + 2) % upcxx::rank_n();
assert(world_rank == expected_world_rank);

upcxx::intrank_t other_local_rank = new_team.from_world(world_rank);
assert(other_local_rank == (local_rank+1)%local_count);
upcxx::intrank_t non_member_rank =
  new_team.from_world((upcxx::rank_me()+1)%upcxx::rank_n(),-1);
assert(non_member_rank == -1);

new_team.destroy(); // collectively release the sub-team
```

In addition to the `upcxx::world()` team, another special team is available that represents all the processes sharing the same shared-memory node. This team is obtained by calling the `upcxx::local_team()` function. All members of this team will see `upcxx::global_ptr<T>` to shared memory allocated by a member of the `local_team` report `is_local() == true`, and `local()` will return a valid raw pointer to the memory. This is particularly important if one wants to optimize for *shared memory* operations.

14 Collectives

Collectives are intimately tied to the notion of Teams. Collectives in UPC++ all operate on the `upcxx::world()` team by default, and always have to be initiated by a thread holding the *master persona* on each process (See Personas). As in other programming models, such as MPI, each collective call in a UPC++ program must be initiated in the same order (with compatible arguments) in every participating process.

One of the most useful and basic collective operations in parallel programming is the *barrier* operation. UPC++ provides two flavors of barrier:

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

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

The first variant is a blocking barrier on `team` and will return only after all processes in the team have entered the call.

The second variant is an asynchronous barrier which by default returns a future (See Completions). This future is signaled when all processes in the team have initiated the asynchronous barrier.

Another fundamental collective operation is the *broadcast* operation, which is always an asynchronous operation in UPC++ and defaults to returning a `upcxx::future`. There are two variants:

```
template <typename T, typename Completions=decltype(upcxx::operation_cx::as_future())>
RType upcxx::broadcast(T &&value, upcxx::intrank_t root,
                       upcxx::team &team = upcxx::world(), Completions cxs=Completions{});

template <typename T, typename Completions=decltype(upcxx::operation_cx::as_future())>
RType upcxx::broadcast(T *buffer, std::size_t count, upcxx::intrank_t root,
                       upcxx::team &team = upcxx::world(), Completions cxs=Completions{});
```

The first variant transfers an object of type `T` from the process with (team-relative) rank `root` to all processes in `team`. The return value is a `upcxx::future<T>` containing the broadcast value. The second variant transfers `count` objects of type `T` stored in an array referenced by `buffer`, and its default return value is an object of type `upcxx::future<>`.

These functions can be used in the following way:

```
int my_rank = upcxx::rank_me();
// launch a first broadcast operation from rank 0
upcxx::future<int> fut = upcxx::broadcast(my_rank, 0);

// do some overlapped work like preparing a buffer for another broadcast
std::vector<int> buffer(10);
if ( upcxx::rank_me() == 0 )
  for (int i = 0; i< buffer.size(); i++)
    buffer[i] = i;

// launch a second broadcast operation from rank 0
upcxx::future<> fut_bulk = upcxx::broadcast( buffer.data(), buffer.size(), 0);

// wait for the result from the first broadcast
int bcast_rank = fut.wait();
assert(bcast_rank == 0);

// wait until the second broadcast is complete
fut_bulk.wait();
for (int i = 0; i< buffer.size(); i++)
  assert(buffer[i] == i);
```

UPC++ also provides collectives to perform *reductions*. Each of these variants applies a binary operator `op` to the input data. This operator can either be one of the built-in operators provided by the library, or can be a user-defined function (for instance a lambda).

```
template <typename T, typename BinaryOp ,
                typename Completions=decltype(upcxx::operation_cx::as_future())>
```

```
RType upcxx::reduce_one(T &&value, BinaryOp &&op, upcxx::intrank_t root,
            upcxx::team &team = upcxx::world(), Completions cxs=Completions{});

template <typename T, typename BinaryOp,
                  typename Completions=decltype(upcxx::operation_cx::as_future())>
RType upcxx::reduce_all(T &&value, BinaryOp &&op,
            upcxx::team &team = upcxx::world(), Completions cxs=Completions{});

template <typename T, typename BinaryOp,
                  typename Completions=decltype(upcxx::operation_cx::as_future())>
RType upcxx::reduce_one(const T *src, T *dst, size_t count, BinaryOp &&op,
            upcxx::intrank_t root, upcxx::team &team = upcxx::world(),
                                                Completions cxs=Completions{});

template <typename T, typename BinaryOp,
                      typename Completions=decltype(upcxx::operation_cx::as_future())>
RType upcxx::reduce_all(const T *src, T *dst, size_t count, BinaryOp &&op,
            upcxx::team &team = upcxx::world(), Completions cxs=Completions{});
```

Similar to `upcxx::broadcast`, the first two variants reduce an object `value` of type `T` and by default return a `upcxx::future<T>` containing the resulting value. The second set of variants perform similar operations on contiguous buffers of `count` objects of type `T` pointed by `src`. They place the reduced values in a contiguous buffer pointed by `dst`, and default to returning a `upcxx::future<>` for tracking completion.

When using the `upcxx::reduce_one` functions, the reduction result will be available only at the `root` process, and the result is undefined on other processes. When using the `upcxx::reduce_all` functions, a reduction result will be provided to all processes belonging to `team`.

UPC++ provides built-in reduction operators (`op_fast_add`, `op_fast_mul`, `op_fast_min`, `op_fast_max`, `op_fast_bit_and`, `op_fast_bit_or`, and `op_fast_bit_xor`) which may be hardware-accelerated on systems with collectives offload support.

15 Non-Contiguous One-Sided Communication

The `rput` and `rget` operations assume a contiguous buffer of data at the source and destination processes. There are specific specialized forms of these functions for moving groups of buffers in a single UPC++ call. These functions are denoted by a suffix `[rput,rget]_[irregular,regular,strided]`. The `rput_irregular` operation is the most general: it takes a set of buffers at the source and destination where the total data size of the combined buffers and their element data type need to match on both ends. The `rput_regular` operation is a specialization of `rput_irregular`, where every buffer in the collection is the same size. The `rput_strided` operation is yet even more specialized: there is just one base address specified for each of the source and destination regions, together with stride vectors describing a multi-dimensional array transfer.

```
constexpr int sdim[] = {32, 64, 32};
constexpr int ddim[] = {16, 32, 64};

upcxx::future<> rput_strided_example(float* src_base, upcxx::global_ptr<float> dst_base)
{
  return upcxx::rput_strided<3>(
      src_base, {{sizeof(float), sdim[0]*sizeof(float), sdim[0]*sdim[1]*sizeof(float)}},
      dst_base, {{sizeof(float), ddim[0]*sizeof(float), ddim[0]*ddim[1]*sizeof(float)}},
      {{4, 3, 2}});
}
```

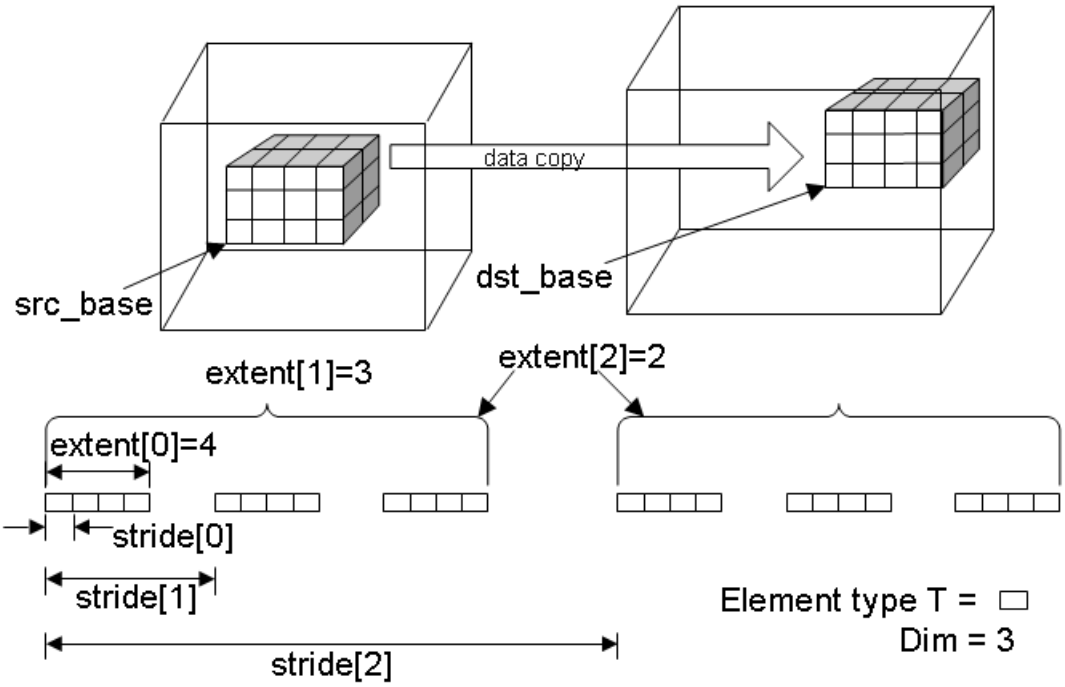

Figure 3: *Figure 3. Example of strided one-sided communication.*

```
upcxx::future<> rget_strided_example(upcxx::global_ptr<float> src_base, float* dst_base)
{
  return upcxx::rget_strided<3>(
      src_base, {{sizeof(float), sdim[0]*sizeof(float), sdim[0]*sdim[1]*sizeof(float)}},
      dst_base, {{sizeof(float), ddim[0]*sizeof(float), ddim[0]*ddim[1]*sizeof(float)}},
      {{4, 3, 2}});
}
```

The strided example code snippet corresponds to the translation data motion shown in Figure 3. By altering the arguments to the `_strided` functions a user can implement various transpose and reflection operations in an N-dimensional space within their UPC++ call. Note the strides are expressed in units of bytes; this enables users to work with data structures that include padding.

For more general data structures a user can use the `_irregular` functions:

```
pair<particle_t*, size_t> src[]={{srcP+12, 22}, {srcP+66, 12}, {srcP, 4}};
pair<upcxx::global_ptr<particle_t>, size_t> dest[]={{destP, 38}};
auto f = upcxx::rput_irregular(src, end(src), dest, end(dest));
f.wait();
```

The user here is taking subsections of their source array of their `particle_t` type pointed to by `srcP` and copying them to the location pointed to by `destP`. All variants of rput and rget assume the element type is TriviallySerializable. This example also shows how data can be shuffled in their UPC++ call. The user is responsible for ensuring their source and destination result in equivalent amounts of data.

16 View-Based Serialization

RPC's transmit their arguments over the wire using the concept of "serialization", which is type-specific logic of how to encode potentially rich C++ objects to and from flat byte buffers. For substantially large objects, deserializing them from UPC++'s internal network buffers can have non-trivial performance cost, and, if the object was built up only to be consumed and immediately torn down within the RPC, then it's likely that performance can be regained by eliminating the superfluous build-up/tear-down. Notably this can happen with containers: Suppose process `A` would like to send a collection of values to process `B` which will assimilate them into its local state. If process A were to transmit these values by RPC'ing them in a `std::vector<T>` (which along with many other `std::` container types, is supported as serializable in the UPC++ implementation) then upon receipt of the RPC, the UPC++ program would enact the following steps:

1. UPC++ would build the vector by visiting each `T` in the network buffer and copying it into the vector.
2. UPC++ would invoke the RPC callback function giving it the vector.
3. The RPC callback function would traverse the `T`'s in the vector and consume them, likely by copying them out to the process's private state.
4. The RPC function would return control to the UPC++ progress engine which would destruct the vector.

This process works, but can entail alot of unnecessary overhead that might be problematic in performance-critical communication. The remedy to eliminating steps 1 and 4 is to allow the application to get direct access to the `T` elements in the internal network buffer. UPC++ grants such access with the `upcxx::view<T>` type. A view is little more than a pair of iterators delimiting the beginning and end of an ordered sequence of `T` values. Since a view only stores iterators, it is not responsible for managing the resources supporting those iterators. Most importantly, when being serialized, a view will serialize each `T` it encounters in the sequence, and when deserialized, the view will "remember" special network buffer iterators delimiting its contents directly in the incoming buffer. The RPC can then ask the view for its begin/end iterators and traverse the `T` sequence in-place.

The following example demonstrates how a user could easily employ views to transmit an array of `double`'s with minimal intermediate copies. On the sender side, the user acquires begin and end iterators to the value sequence they wish to send (in this case `double*` acts as the iterator type) and calls `upcxx::make_view(beg,end)` to construct the view. That view is bound to an `rpc` whose lambda accepts a `upcxx::view<double>` on the receiver side, and traverses the view to consume the sequence.

```
// Simple demonstration of shipping an array of doubles by view.
double *buf = /*...*/; // local data buffer
size_t len = /*...*/;
upcxx::rpc(target_rank,
  [](upcxx::view<double> buf_in_rpc) {
    // Traverse `buf_in_rpc` like a container. Views fulfill most of the
    // container contract: begin, end, size, and if the element type is
    // trivial, even operator[].
    for(double x: buf_in_rpc)
      /* consume each `x` */;
  },
  upcxx::make_view(buf, buf + len)
);
```

Beyond just simple pointers to contiguous data, arbitrary iterator types can be used to make a view. This allows the user to build views from the sequence of elements within `std` containers using `upcxx::make_view(container)`, or, given any compliant `ForwardIterator`, `upcxx::make_view(begin_iter, end_iter)`.

For a more involved example, we will demonstrate one process contributing histogram values to a histogram distributed over all the processes. We will use `std::string` as the key-type for naming histogram buckets,

`double` for the accumulated bucket value, and `std::unordered_map` as the container type for mapping the keys to the values. Assignment of bucket keys to owning process is done by a hash function. We will demonstrate transmission of the histogram update with and without view's, illustrating the performance advantages that views enable.

```
// Hash a key to its owning rank.
upcxx::intrank_t owner_of(std::string const &key) {
  std::uint64_t h = 0x1234abcd5678cdef;
  for(char c: key)
    h = 63*h + std::uint64_t(c);
  return h % upcxx::rank_n();
}

using histogram1 = std::unordered_map<std::string, double>;

// The target rank's histogram which is updated by incoming rpc's.
histogram1 my_histo1;

// Sending histogram updates by value.
upcxx::future<> send_histo1_byval(histogram1 const &histo) {
  std::unordered_map<upcxx::intrank_t, histogram1> clusters;

  // Cluster histogram elements by owning rank.
  for(auto const &kv: histo)
    clusters[owner_of(kv.first)].insert(kv);

  upcxx::promise<> *all_done = new upcxx::promise<>;

  // Send per-owner histogram clusters.
  for(auto const &cluster: clusters) {
    upcxx::rpc(cluster.first,
      upcxx::operation_cx::as_promise(*all_done),

      [](histogram1 const &histo) {
        // Pain point: UPC++ already traversed the key-values once to build the
        // `histo` container. Now we traverse again within the RPC body.

        for(auto const &kv: histo)
          my_histo1[kv.first] += kv.second;

        // Pain point: UPC++ will now destroy the container.
      },
      cluster.second
    );
  }

  return all_done->finalize().then(
    [=]() { delete all_done; }
  );
}

// Sending histogram updates by view.
upcxx::future<> send_histo1_byview(histogram1 const &histo) {
  std::unordered_map<upcxx::intrank_t, histogram1> clusters;
```

```
  // Cluster histogram elements by owning rank.
  for(auto const &kv: histo)
    clusters[owner_of(kv.first)].insert(kv);

  upcxx::promise<> *all_done = new upcxx::promise<>;

  // Send per-owner histogram clusters.
  for(auto const &cluster: clusters) {
    upcxx::rpc(cluster.first,
      upcxx::operation_cx::as_promise(*all_done),

      [](upcxx::view<std::pair<const std::string, double>> histo_view) {
        // Pain point from `send_histo1_byval`: Eliminated.

        // Traverse key-values directly in network buffer.
        for(auto const &kv: histo_view)
          my_histo1[kv.first] += kv.second;

        // Pain point from `send_histo1_byval`: Eliminated.
      },
      upcxx::make_view(cluster.second) // build view from container's begin()/end()
    );
  }

  return all_done->finalize().then(
    [=]() { delete all_done; }
  );
}
```

There is a further benefit to using view-based serialization: the ability for the sender to serialize a subset of elements directly out of a container without preprocessing it (as is done in the two examples above). This is most efficient if we take care to use a container that natively stores its elements in an order grouped according to the destination process. The following example demonstrates the same histogram update as before, but with a data structure that permits sender-side subset serialization.

```
// Hash a key to its owning rank.
upcxx::intrank_t owner_of(std::string const &key) {
  std::uint64_t h = 0x1234abcd5678cdef;
  for(char c: key)
    h = 63*h + std::uint64_t(c);
  return h % upcxx::rank_n();
}

// This comparison functor orders keys such that they are sorted by
// owning rank at the expense of rehashing the keys in each invocation.
// A better strategy would be modify the map's key type to compute this
// information once and store it in the map.
struct histogram2_compare {
  bool operator()(std::string const &a, std::string const &b) const {
    using augmented = std::pair<upcxx::intrank_t, std::string const&>;
    return augmented(owner_of(a), a) < augmented(owner_of(b), b);
  }
};
```

```
using histogram2 = std::map<std::string, double, histogram2_compare>;

// The target rank's histogram which is updated by incoming rpc's.
histogram2 my_histo2;

// Sending histogram updates by view.
upcxx::future<> send_histo2_byview(histogram2 const &histo) {
  histogram2::const_iterator run_begin = histo.begin();

  upcxx::promise<> *all_done = new upcxx::promise<>;

  while(run_begin != histo.end()) {
    histogram2::const_iterator run_end = run_begin;
    upcxx::intrank_t owner = owner_of(run_begin->first);

    // Compute the end of this run as the beginning of the next run.
    while(run_end != histo.end() && owner_of(run_end->first) == owner)
      ++run_end;

    upcxx::rpc(owner,
      upcxx::operation_cx::as_promise(*all_done),

      [](upcxx::view<std::pair<const std::string, double>> histo_view) {
        // Traverse key-values directly in network buffer.
        for(auto const &kv: histo_view)
          my_histo2[kv.first] += kv.second;
      },
      // Serialize from a subset of `histo` in-place.
      upcxx::make_view(run_begin, run_end)
    );

    run_begin = run_end;
  }

  return all_done->finalize().then(
    [=]() { delete all_done; }
  );
}
```

16.1 The `view`'s Iterator Type

The above text presented correct and functional code, but it oversimplified the C++ type of the UPC++ view by relying on some of its default characteristics and type inference. The full type signature for view is:

```
upcxx::view<T, Iter=/*internal buffer iterator*/>
```

Notably, the view type has a second type parameter which is the type of its underlying iterator. If omitted, this parameter defaults to a special UPC++ provided iterator that deserializes from a network buffer, hence this is the correct type to use when specifying the incoming bound-argument in the RPC function. But this will almost never be the correct type for the view on the sender side of the RPC. For instance, `upcxx::make_view(...)` deduces the iterator type provided in its arguments as the `Iter` type to use in the returned view. If you were to attempt to assign that to an temporary variable you might be surprised:

```
std::vector<T> vec = /*...*/;
```

```
// Type error: mismatched Iter types
upcxx::view<T> tmp = upcxx::make_view(vec);

// OK: tmp deduced to: upcxx::view<T, std::vector<T>::const_iterator>
auto tmp = upcxx::make_view(vec);
```

16.2 Buffer Lifetime Extension

Given that the lifetime of a view does not influence the lifetime of its underlying data, UPC++ must make guarantees to the application about the lifetime of the network buffer when referenced by a view. From the examples above, it should be clear that UPC++ will ensure the buffer will live for at least as long as the RPC function is executing. In fact, UPC++ will actually extend this lifetime until the future (if any) returned by the RPC is ready. This gives the application a convenient means to dispatch the processing of the view to another concurrent execution agent (e.g. a thread), thereby returning from the RPC nearly immediately so that UPC++ can service more progress events.

The following example demonstrates how a process can send sparse updates to a remote matrix via `rpc`. The updates are not done in the execution context of the `rpc` itself, instead the `rpc` uses `lpc`'s to designated worker personas (backed by dedicated threads) to dispatch the arithmetic update of the matrix element depending on which worker owns it. Views and futures are used to extend the lifetime of the network buffer until all `lpc`'s have completed, thus allowing those `lpc`'s to use the elements directly from the buffer.

```
// Number of worker threads/personas.
constexpr int worker_n = 8;

// Each persona has a dedicated thread spinning on its progress engine.
upcxx::persona workers[worker_n];

// Rank's local matrix.
double my_matrix[1000][1000] = {/*0...*/};

struct element {
  int row, col;
  double value;
};

upcxx::future<> update_remote_matrix(
    upcxx::intrank_t rank,
    element const *elts, int elt_n
  ) {
  return upcxx::rpc(rank,
    [](upcxx::view<element> elts_in_rpc) {
      upcxx::future<> all_done = upcxx::make_future();

      for(int w=0; w < worker_n; w++) {
        // Launch task on respective worker.
        auto task_done = workers[w].lpc(
          [w, elts_in_rpc]() {
            // Sum subset of elements into `my_matrix` according to a
            // round-robin mapping of matrix rows to workers.
            for(element elt: elts_in_rpc) {
              if(w == elt.row % worker_n)
                my_matrix[elt.row][elt.col] += elt.value;
            }
```

```
          }
        );

        // Conjoin task completion into `all_done`.
        all_done = upcxx::when_all(all_done, task_done);
      }

      // Returned future has a dependency on each task lpc so the network
      // buffer (thus `elts_in_rpc` view) will remain valid until all tasks
      // have completed.
      return all_done;
    },
    upcxx::make_view(elts, elts + elt_n)
  );
}
```

17 Memory Kinds

The memory kinds interface enables the UPC++ 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. UPC++ currently handles one new memory kind – GPU memory in NVIDIA-branded CUDA devices – but in the future it may be extended to handle other types of memory.

We demonstrate how to use memory kinds to transfer data between device and host memories, and then between host and device residing on different nodes. For simplicity our examples assume a single device per node, but UPC++ can also handle nodes with heterogeneous device counts. See the *UPC++ Specification* for the details.

In our first example, we allocate a block of storage in device and host memories and then move the data from host to GPU.

To allocate storage, we first open a device CUDA Device IDs start at 0; since we have only one device in this example, we open device 0:

```
upcxx::init();
auto gpu_device = upcxx::cuda_device( 0 );   // Open device 0
```

The next step is to construct an allocator, which we then use to allocate the actual storage. Of note, we pass the segment size to the allocator– in our case 4MB. If we need a larger segment, we need to specify that at the time we construct the allocator. Currently, there is no way to dynamically extend the segment.

```
std::size_t segsize = 4*1024*1024;          // 4MB
// Allocate GPU segment
auto gpu_alloc = upcxx::device_allocator<upcxx::cuda_device>(gpu_device, segsize);
```

The next step is to allocate an array of 1024 doubles on the GPU, by calling the `allocate` function of the allocator we have just created.

```
// Allocate an array of 1024 doubles on GPU
global_ptr<double,memory_kind::cuda_device> gpu_array = gpu_alloc.allocate<double>(1024);
```

Next, we allocate the host storage using the familiar `new_array` method:

```
global_ptr<double> host_array = upcxx::new_array<double>(1024);
```

Data movement is handled by the `upcxx::copy()` function. We transfer 1024 doubles from the host buffer to the GPU buffer. As with `rput` and `rget`, `copy` is an asynchronous operation that returns a future. In this case, we simply use wait, but the full machinery of Completions is available (see the *Programmer's Guide* for the details).

```
upcxx::copy(host_array, gpu_array, 1024).wait();
```

If we wanted to move the data in the opposite direction, we'd simply swap the `host_array` and `gpu_array` arguments. Note the absence of explicit CUDA data movement calls.

After transferring the data, most users will need to invoke a computational kernel on the device, and will therefore need a raw pointer to the data allocated on the device. The `device_allocator::allocate` function returns a `global_ptr` to the data in the specific segment associated with that allocator. This `global_ptr` can be downcast to a raw device pointer using the allocator's `local` function.

```
template<typename Device>
template<typename T>
Device::pointer<T> device_allocator<Device>::local(global_ptr<T, Device::kind> g);
```

This function will return the raw device pointer which can then be used as an argument to a CUDA computational kernel.

In a similar fashion, a raw device pointer into the segment managed by a `device_allocator` may be upcast into a `global_ptr` using the `to_global_ptr` function of that allocator.

```
template<typename Device>
template<typename T>
global_ptr<T, Device::kind> device_allocator<Device>::to_global_ptr(Device::pointer<T> ptr);
```

In the case of CUDA, it is important to obtain the device id in order to be able to launch a kernel. This can be retrieved using the `cuda_device::device_id` method.

Here is a complete example:

```
#include <upcxx/upcxx.hpp>
#include <iostream>
using namespace std;
using namespace upcxx;

#if !UPCXX_CUDA_ENABLED
#error "This example requires UPC++ to be built with CUDA support."
#endif

int main() {
  upcxx::init();

  std::size_t segsize = 4*1024*1024; // 4MB
  auto gpu_device = upcxx::cuda_device( 0 ); // open device 0
  auto gpu_alloc = // alloc GPU segment
       upcxx::device_allocator<upcxx::cuda_device>(gpu_device, segsize);

  // alloc an array of 1024 doubles on GPU and host
  global_ptr<double,memory_kind::cuda_device> gpu_array = gpu_alloc.allocate<double>(1024);
  global_ptr<double> host_array1 = new_array<double>(1024);
  global_ptr<double> host_array2 = new_array<double>(1024);

  double *h1 = host_array1.local();
```

```
  double *h2 = host_array2.local();
  // initialize h1
  for (int i=0; i< 1024; i++)
    h1[i] = i;

  // copy data from host memory to GPU
  upcxx::copy(host_array1, gpu_array, 1024).wait();
  // copy data back from GPU to host memory
  upcxx::copy(gpu_array, host_array2, 1024).wait();

  int nerrs = 0;
  for (int i=0; i< 1024; i++){
    if (h1[i] != h2[i]){
      if (nerrs < 10)
        cout << "Error at element " << i << endl;
      nerrs++;
    }
  }
  if (nerrs)
    cout << "Failure: " << nerrs << " errors detected\n";
  else
    cout << "Success" << endl;

  delete_array(host_array2);
  delete_array(host_array1);
  gpu_alloc.deallocate(gpu_array);

  gpu_device.destroy();
  upcxx::finalize();
}
```

The power of the UPC++ memory kinds facility lays in its unified interface for moving data not only between host and device on the same node, but between source and target that reside on different nodes. As in the previous example, the copy method leverages UPC++'s global pointer abstraction. We next show how to move data between a host and device that reside on different nodes.

Assuming that we have set up a GPU allocator, we allocate host and device buffers as before. We use `dist_object` as a directory for fetching remote global pointers, which we can then use in the `copy()` method:

```
  dist_object<global_ptr<double,memory_kind::cuda_device>> dobj(gpu_array);
  int neighbor = (rank_me() + 1) % rank_n();
  global_ptr<double,memory_kind::cuda_device> other_gpu_array = dobj.fetch(neighbor).wait();

  // copy data from local host memory to remote GPU
  upcxx::copy(host_array1, other_gpu_array, 1024).wait();
  // copy data back from remote GPU to local host memory
  upcxx::copy(other_gpu_array, host_array2, 1024).wait();

  upcxx::barrier();
```

Again, if we want to reverse the direction of the data motion, we need only swap the first two arguments in the call to `copy()`.

Our code concludes by deallocating host and node buffers. Device allocator has its own method for deallocation:

```
  gpu_alloc.deallocate(gpu_array);
  upcxx::delete_array(host_array);
```

Note that in this second example, we have added a call to `upcxx::barrier` to prevent a process from deallocating its segment while another rank is copying data (See Quiescence).

In this tutorial, we've shown how to use the memory kinds in UPC++. This powerful feature relies on a unified notion of the global pointer abstraction to enable methods that use the pointer to interpret the type of memory the pointer refers to, while hiding the details of the actual access method from the user.

We've not demonstrated how to move data directly between GPUs. We leave the details as an exercise to the reader.