Lawrence Berkeley National Laboratory

Title
UPC++ v1.0 Programmer's Guide, Revision 2023.3.0

Permalink
https://escholarship.org/uc/item/5c7729r0

Authors

Bachan, John
Baden, Scott B
Bonachea, Dan
et al.

UPC++ v1.0 Programmer's Guide
Revision 2023.3.0

Lawrence Berkeley National Laboratory Technical Report (LBNL-2001517)

Contents

1   Introduction

UPC++ is a C++ library that supports Partitioned Global Address Space (PGAS) programming. It is designed for writing efficient, scalable parallel programs on distributed-memory parallel computers. The key communication facilities in UPC++ are one-sided Remote Memory Access (RMA) and Remote Procedure Call (RPC). The UPC++ control model is single program, multiple-data (SPMD), with each separate constituent process having access to local memory as it would in C++. The PGAS memory model additionally provides one-sided RMA communication to a global address space, which is allocated in shared segments that are distributed over the processes (see Figure 1). UPC++ also features Remote Procedure Call (RPC) communication, making it easy to move computation to operate on data that resides on remote processes.

UPC++ was designed to support exascale high-performance computing, and the library interfaces and implementation are focused on maximizing scalability. In UPC++, all communication operations are syntactically explicit, which encourages programmers to consider the costs associated with communication and data movement. Moreover, all communication operations are asynchronous by default, encouraging programmers to seek opportunities for overlapping communication latencies with other useful work. UPC++ provides expressive and composable abstractions designed for efficiently managing aggressive use of asynchrony in programs. Together, these design principles are intended to enable programmers to write applications using UPC++ that perform well even on hundreds of thousands of cores.
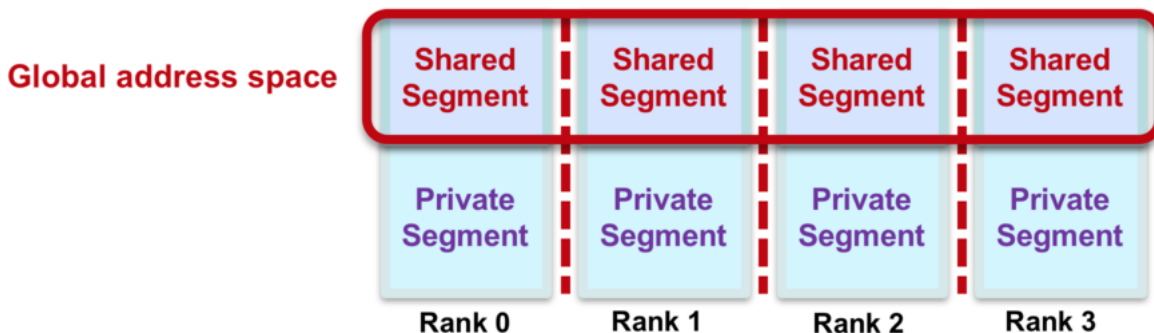
Figure 1: *PGAS Memory Model.*

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++, Fortran coarrays, 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++ and Advanced Job Launch sections are specific to the implementation. The LBNL implementation of UPC++ adheres to the implementation-independent UPC++ Specification. Both are available at the UPC++ homepage at https://upcxx.lbl.gov/.

See Additional Resources for links to additional sites that are helpful in learning to use UPC++. For example, the UPC++ training site at https://upcxx.lbl.gov/training has links to introductory video tutorials that give overviews of using UPC++ and some hands-on exercises to get started.

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 C++ templates, the C++ standard library (`std::`), and lambda functions, which are used heavily in this guide.

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 and README.md files that come with the distribution.

2.1   Installing UPC++

UPC++ installation follows the same steps as many GNU autotools-based projects: `configure`, `make all`, `make install`. To install UPC++, extract the library source code to a clean directory, and from the top-level of this directory, run the following steps (output has been omitted for brevity):

```
$ ./configure --prefix=<upcxx-install-path>
$ make all
$ make install
```

This will configure, build and install the UPC++ library in the chosen `<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>`.

Optionally, `make check` (after `make all`) and `make test_install` (after `make install`) can be run to validate the build and installation steps.

2.1.1   General Requirements

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

2.1.2   Requirements for Cray XC

UPC++ supports Cray XC systems, with native communication support provided by the `aries` network backend. If a Cray `PrgEnv-*` environment module is loaded at the time the `configure` script is run, it will automatically utilize the `cc` and `CC` compiler wrappers of the loaded Cray programming environment. This ensures that UPC++ installation is built to target the XC compute nodes. It will also configure `upcxx-run` to use either the Cray ALPS or SLURM job launcher. If detection of the appropriate job launcher fails, you may need to pass a `--with-cross=...` option to the `configure` command.
More details are provided in INSTALL.md.

2.1.3   Requirements for HPE Cray EX

UPC++ supports HPE Cray EX (aka Shasta) systems, with native communication support provided by the `ofi` network backend. Please see INSTALL.md for details on configuring UPC++ for HPE Cray EX.

2.1.4   Requirements for macOS

To install UPC++ on macOS, the Xcode Command Line Tools must be installed *before* invoking `configure`, i.e.:

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

2.2   Compiling UPC++ Programs

To compile a program using UPC++, the `<upcxx-install-path>/bin/upcxx` wrapper can be used. This works analogously to the familiar `mpicxx` wrapper, in that it invokes the underlying C++ compiler with the provided options, and automatically prepends the options necessary for including/linking the UPC++ library. For example to build the hello world code shown in an upcoming section, one could execute:

```
<upcxx-install-path>/bin/upcxx -O3 hello-world.cpp
```

Note that an option starting either with `-O` (for optimized) or `-g` (for debug mode) should be specified, which has a side-effect of selecting the variant of the UPC++ library built with optimization (for production runs) or with assertions and debugging symbols (for debugging and development). Alternatively, one can effect the same choice by passing `upcxx -codemode={opt,debug}` The debug-mode library is highly recommended for all application development, as it includes many useful correctness assertions that can help reveal application-level bugs. Conversely, the opt-mode library should be used for all performance testing or production runs. Note that most `upcxx` options are passed straight through to the C++ compiler, which includes options controlling the optimization level (if any). If no `-O` options are provided (or only a bare "`-O`"), the wrapper provides a compiler-specific default optimization level that might not be ideal for your program (selecting the best optimization options for your C++ compiler is application-specific and beyond the scope of this guide).

If features from a C++ standard beyond C++11 are required, the C++ standard may also need to be specified (for example, by passing `-std=c++14` or `-std=c++17`). For a complete 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++ supports multithreading within a process, e.g. using OpenMP. In these cases, to ensure that the application is compiled against a thread-safe UPC++ library, pass `upcxx -threadmode=par`. Note that this option is less efficient than the default, `upcxx -threadmode=seq`, which enables the use of a UPC++ library that is synchronization-free in most of its internals; thus, the parallel, thread-safe mode should only be used when multithreading within processes. For detailed restrictions associated with `-threadmode=seq`, consult the `docs/implementation-defined.md` document provided by the distribution.

UPC++'s communication services are implemented over GASNet, and on a given system there may be several communication backends available corresponding to different network hardware and/or software stacks. The communication backend can be selected using `upcxx -network=<name>`.

For example, to select the debug, thread-safe version of the UDP communication backend:

```
<upcxx-install-path>/bin/upcxx -g -network=udp -threadmode=par hello-world.cpp
```

Environment variables are available to establish defaults for several of the common options. The variables `UPCXX_CODEMODE`, `UPCXX_THREADMODE` and `UPCXX_NETWORK` provide defaults for `-codemode`, `-threadmode` and `-network`, respectively.

There are several other options that can be passed to `upcxx`. Execute `upcxx --help` to get the full list of options, e.g.:

```
upcxx is a compiler wrapper that is intended as a drop-in replacement for your
C++ compiler that appends the flags necessary to compile/link with the UPC++ library.
Most arguments are passed through without change to the C++ compiler.
Citing UPC++ in publication? Please see: https://upcxx.lbl.gov/publications


Usage: upcxx [options] file...

upcxx Wrapper Options:
---------------------


 Informational queries (no compilation):
  -help          This message
  -version       Print UPC++ and compiler version information
  -info          Print full configuration info

 UPC++ library configuration:
  -network={ibv|aries|ofi|ucx|smp|udp|mpi}
                    Use the indicated GASNet network backend for communication.
```

```
                   The default and availability of backends is system-dependent.
   -codemode={opt|debug}
                   Select the optimized or debugging variant of the UPC++ library.
   -threadmode={seq|par}
                   Select the single-threaded or thread-safe variant of the UPC++ library.


 All other C++ compiler options:
   -Wc,<anything>   <anything> is passed-through uninterpreted to the underlying compiler
   <anything-else>  Passed-through uninterpreted to the underlying compiler

C++ compiler --help:
Usage: g++ [options] file...
[...snip...]
```

More details about compilation can be found in the README.md file that comes with the distribution.

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 `upcxx-run --help` to get the full list of options, e.g.:

```
usage: upcxx-run [-h] [-n NUM] [-N NUM] [-shared-heap HEAPSZ] [-backtrace] [-show]
                 [-info] [-ssh-servers HOSTS] [-localhost] [-v] [-E VAR1[,VAR2]] [--]
                 command ...


A portable parallel job launcher for UPC++ programs

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: don't 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                  Generate verbose output. Multiple applications increase
                      verbosity.
  -E VAR1[,VAR2]      Adds provided arguments to the list of environment variables
```

```
                        to propagate to compute processes.
  --                    Optional separator between upcxx-run arguments and command to
                        execute

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

Use of `-v`, `-vv` and `-vvv` provide additional information which may be useful. Among other things, `-v` (or higher) will print job layout and shared-heap usage information.

The `upcxx-run` utility covers the basic usage cases, but does not have options to cover all possibilities, such as establishing core and memory affinity. However, `upcxx-run` is just a wrapper around system-provided job launch tools with their associated options. For information on use of system-provided job launch tools with UPC++ applications see Advanced Job Launch.

2.4 UPC++ preprocessor defines

The header `upcxx.hpp` defines certain preprocessor macros to reflect the version and compilation environment:

2.4.1 `UPCXX_VERSION`

An integer literal providing the release version of the implementation, in the format `[YYYY][MM][PP]` corresponding to release `YYYY.MM.PP`.

2.4.2 `UPCXX_SPEC_VERSION`

An integer literal providing the revision of the UPC++ specification to which this implementation adheres. See the specification for its corresponding value.

2.4.3 `UPCXX_THREADMODE`

This is either undefined (for the default "seq" threadmode) or defined to an unspecified non-zero integer value for the "par" threadmode. Recommended usage is to test using `#if`, as follows, to identify the need for thread-safety:

```
#if UPCXX_THREADMODE
   // Obtain a lock or do other thread-safety work
#endif
```

2.4.4 `UPCXX_CODEMODE`

This is either undefined (for the "debug" codemode) or defined to an unspecified non-zero integer value for the "opt" (production) codemode.

2.4.5 `UPCXX_NETWORK_*`

The network being targeted is indicated by a preprocessor macro with a `UPCXX_NETWORK_` prefix followed by the network name in capitals. The macro for the network being targeted is defined to a non-zero integer value, while those corresponding to others network are undefined.

Currently supported networks and their macros include:

Network/System	Macro
InfiniBand OpenIB/OpenFabrics Verbs	UPCXX_NETWORK_IBV
Aries (Cray XC)	UPCXX_NETWORK_ARIES
OFI Libfabric (HPE Cray EX, Intel Omni-Path)	UPCXX_NETWORK_OFI
UDP (e.g. for Ethernet)	UPCXX_NETWORK_UDP
No network (single node)	UPCXX_NETWORK_SMP

3 Hello World in UPC++

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

```cpp
#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;
}
```

(see full example online)

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:

```cpp
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`) which points to an integer object in its local shared segment. By contrast, a conventional C++ dynamic allocation (`int *mine = new int`) allocates an integer object 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.
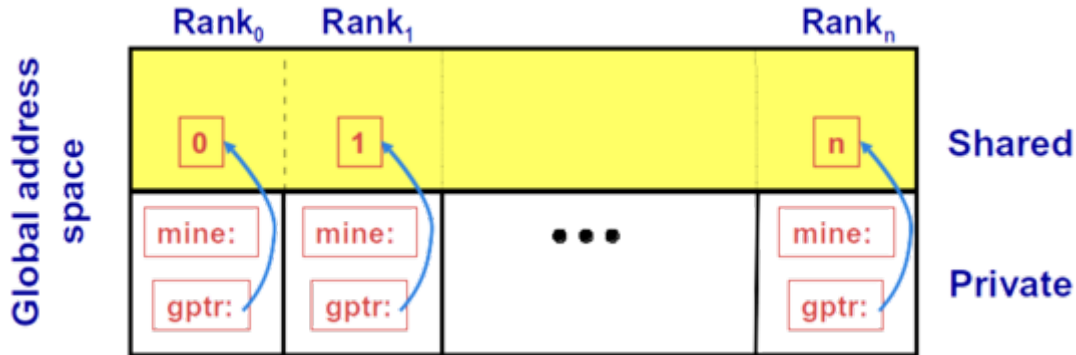
Figure 2: *Global pointers.*

A UPC++ global pointer is fundamentally different from a conventional C++ pointer: it cannot be dereferenced using the `*` operator; it does not support dynamic casts (e.g., polymorphic conversion of pointers between base and derived types); and it cannot be constructed by the C++ `std::addressof` operator. However, UPC++ global pointers do support pointer arithmetic and are trivially copyable.

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);
```

UPC++ provides the function `upcxx::new_array` for allocating a 1-dimensional array in the shared segment. This is a non-collective call and the array is NOT distributed across processes. Each call allocates an independent, contiguous array object in the caller's shared segment. `upcxx::new_array` operates analogously to C++ array new and default-initializes objects (meaning it calls the *default* constructor for class types, and leaves other types with indeterminate value). The destruction operation is `upcxx::delete_array`. For example, to allocate a shared array of 10 uninitialized integers and then free it:

```
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);
UPCXX_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
```

UPCXX_ASSERT() is a macro analogous to the C library `assert()` macro from `<cassert>`, except the assertion check is conditionally enabled using `UPCXX_CODEMODE` instead of the `NDEBUG` define. There is also a `UPCXX_ASSERT_ALWAYS()` variant that performs an unconditional check. Both macros accept an optional second argument that can be used to provide a console message to print upon failure, which can be a simple string or a stream insertion expression.

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 incorrect to attempt to downcast a global pointer that is *not* local to a process. In "debug" codemode, such attempts are automatically diagnosed with a fatal error at runtime.

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> gptr_panel = 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. For simplicity, we also assume `n_local` is even.

5.1 Distributed Objects

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 allocate our panels and declare a distributed object to hold the global pointers to the panels:

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

Each process in a given team (in this case the default team containing all processes) collectively calls this constructor 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 this `dist_object` constructor 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 fully constructed and activated.

We can still use `global_ptr::local()` 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 l_nbr = (upcxx::rank_me() + upcxx::rank_n() - 1) % upcxx::rank_n();
int r_nbr = (upcxx::rank_me() + 1) % upcxx::rank_n();
upcxx::global_ptr<double> uL = u_g.fetch(l_nbr).wait();
upcxx::global_ptr<double> uR = u_g.fetch(r_nbr).wait();
```

Because the fetch function is asynchronous, we have to synchronize on completion using a call to `wait()`. In the Asynchronous Computation section, we will see how to overlap asynchronous operations, that is, when communication is split-phased. For now, we do not separate the asynchronous initiation from the `wait()`.

5.2 RMA communication

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 Remote Memory Access (RMA) communication 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 requirements. 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 = 1024;
  const long MAX_ITER = N * N * 2;
  const double EPSILON = 0.1;
  const int MAX_VAL = 100;
  const double EXPECTED_VAL = MAX_VAL / 2;
  // get the bounds for the local panel, assuming num procs divides N into an even block size
  long block = N / upcxx::rank_n();
  UPCXX_ASSERT(block % 2 == 0); UPCXX_ASSERT(N == block * upcxx::rank_n());
  long n_local = block + 2; // plus two for ghost cells
  // 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();
  // init to uniform pseudo-random distribution, independent of job size
  mt19937_64 rgen(1);  rgen.discard(upcxx::rank_me() * block);
  for (long i = 1; i < n_local - 1; i++)
    u[i] = 0.5 + rgen() % MAX_VAL;
  // fetch the left and right pointers for the ghost cells
  int l_nbr = (upcxx::rank_me() + upcxx::rank_n() - 1) % upcxx::rank_n();
  int r_nbr = (upcxx::rank_me() + 1) % upcxx::rank_n();
  upcxx::global_ptr<double> uL = u_g.fetch(l_nbr).wait();
  upcxx::global_ptr<double> uR = u_g.fetch(r_nbr).wait();
  upcxx::barrier(); // optional - wait for all ranks to finish init
  // iteratively solve
  for (long stepi = 0; stepi < MAX_ITER; stepi++) {
    // alternate between red and black
    int phase = stepi % 2;
    // get the values for the ghost cells
    if (!phase) u[0] = upcxx::rget(uL + block).wait();
    else u[n_local - 1] = upcxx::rget(uR + 1).wait();
    // compute updates and error
    for (long i = phase + 1; i < n_local - 1; i += 2)
      u[i] = (u[i - 1] + u[i + 1]) / 2.0;
    upcxx::barrier(); // wait until all processes have finished calculations
    if (stepi % 10 == 0) { // periodically check convergence
      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, long n_local, const double EXPECTED_VAL,
                       const double EPSILON, long stepi)
{
  double err = 0;
  for (long 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;
}
```

(see full example online)

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 currently the function cannot be in a shared library. The arguments to an RPC must be Serializable, generally either by satisfying the C++ TriviallyCopyable requirement (which includes all builtin types and many class types), or be one of the commonly used types from the C++ standard template library. See Serialization for details, including interfaces for defining non-trivial serialization of user-defined types.

6.1   Implementing a Distributed Hash Table

We illustrate the use of RPCs with a simple distributed hash table, a distributed analog of the C++ unordered map container. 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

15

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, const std::string &key, const std::string &val) {
                      lmap->insert({key, val});
                    }, local_map, key, val);
}
```

Note 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`. Also note the use of const-reference for other arguments, to avoid the cost of copying the RPC arguments upon entry to the callback.

The find operation for our distributed map 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, const std::string &key, const 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, const std::string &key) -> std::string {
                        auto elem = lmap->find(key);
                        if (elem == lmap->end()) return std::string(); // not found
                        else return elem->second; // key found: return value
                      }, 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:

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

using namespace std;

int main(int argc, char *argv[])
{
  upcxx::init();
  const long N = 1000;
  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
    UPCXX_ASSERT(val == key);
  }
  upcxx::barrier(); // wait for finds to complete globally
  if (!upcxx::rank_me()) cout << "SUCCESS" << endl;
  upcxx::finalize();
  return 0;
}
```

(see full example online)

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.

6.2   RPC and Lambda Captures

We've seen several RPC examples that specify a remote callback by passing a C++ lambda expression, which is concise and convenient for small blocks of code. RPC actually accepts any form of C++ *FunctionObject* for the callback, which includes global and file-scope functions, class static functions, function pointers, lambda expressions and even user-defined objects implementing `operator()`. RPC includes system-specific logic to automatically ensure these various callables are correctly serialized and invoked at the target process. However it's important to understand that when passing a lambda expression as the FunctionObject, RPC will assume that lambda is trivially copyable and it will be byte-copied when transmitted to the target process. As a result, special care needs to be taken when using lambda captures with RPC.

In particular:

When passing a C++ lambda expression to an RPC for execution on another process, by-reference lambda 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. Note this rule does not apply to LPC and `future::then()` callbacks (discussed in upcoming sections), which always execute in the same process, but the normal C++ precautions regarding reference captures and object lifetimes still apply.

Our examples thus far have used lambda expressions with an empty capture list, i.e.: `[/*empty*/](...) { ... }`, which trivially avoids this pitfall. It *is* possible to safely use *value* captures (also known as capture by-copy) of *trivial* types in a lambda expression passed to RPC, however keep in mind the lambda object (including any captures) are still assumed to be trivially copyable:

> *When passing a C++ lambda expression to an RPC, lambda value captures (capture by-copy) should only include objects which are trivially copyable.*

When in doubt, it's always safer to pass data to an RPC callback (lambda or otherwise) using explicit RPC callback arguments as we've done in our examples so far, instead of relying upon lambda captures or data otherwise embedded in the FunctionObject. In section Serialization we'll learn about the UPC++ machinery that enables RPC callback arguments to transmit rich data types.

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 idiom 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();
}
```

(see full example online)

18

UPC++ also provides *callbacks* or *completion handlers* that can be attached to futures. These are functions that are executed by the local process 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](const string &val) {
      UPCXX_ASSERT(val == key);
    });
  // wait for future and its callback to complete
  fut.wait();
}
```

(see full example online)

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 cumbersome to individually track 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](const string &val) {
      UPCXX_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();
```

(see full example online)

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`

19

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. Figure 3 depicts the dependency graph of operations constructed when this code executes, with each blue box conceptually corresponding to a future constructed synchronously at runtime. The boxes are labeled with a fragment of the code that activates after any/all incoming dependencies (depicted by arrows) are readied, and eventually leads to readying of the future associated with this box. For instance, the callback passed to each `then` will execute at some time after its preceding `dmap.find()` has completed asynchronously. 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).
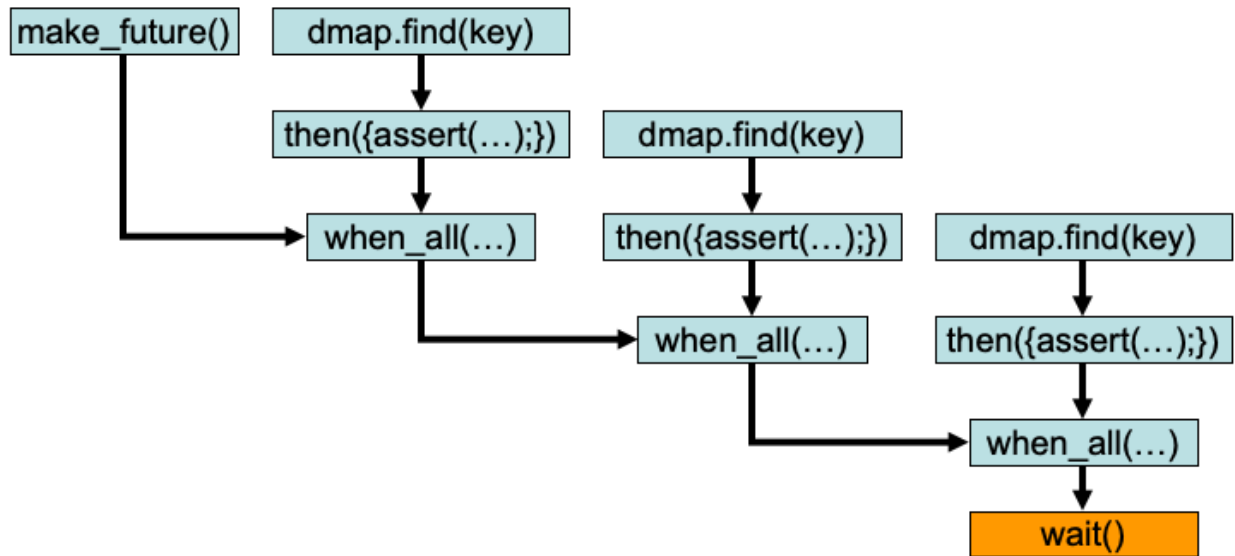
Figure 3: *Graph of future conjoining.*

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.

7.2 Multi-valued Futures and Structured Bindings

We saw above that futures can encapsulate the statuses of multiple asynchronous operations. Additionally, they can encapsulate multiple values. These values are contained in a `std::tuple` that can be accessed using the `result()` or `wait()` member functions of `upcxx::future`. Using the C++11 standard, these values can be accessed as shown below. The `make_future` function is variadic, and constructs a ready future from the arguments.

```
int a;
double b;
upcxx::future<int,double> fut = upcxx::rpc(0,[]() { return upcxx::make_future(42, 3.14); });
std::tie(a,b) = fut.wait();
```

However, the code can be expressed much more concisely using C++17 structured bindings.

```
upcxx::future<int,double> fut = upcxx::rpc(0,[]() { return upcxx::make_future(42, 3.14); });
auto [a,b] = fut.wait();
```

In both of the above cases `a` and `b` are copies of the values contained within `fut`. If copying these values is expensive or unnecessary, `wait_reference()` can be used to obtain `const` references, instead.

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. It's important to note that quiescence is also a precondition for calling `upcxx::finalize()` during an orderly exit of the parallel job.

To illustrate a simple approach to quiescence, we use the distributed hash table again. In this case, we use a variant of RPC that does not return a future, whose simplest overload looks like this:

```
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, `upcxx::rpc_ff` has the advantage that it does not *send a response message* to satisfy a `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 of `func` 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, const std::string &key, const std::string &val) {
                  UPCXX_ASSERT(lmap->count(key) == 0); // assume no duplicate keys
                  // insert into the local map at the target
                  lmap->insert({key, val});
                }, local_map, key, val);
}
```

(see full example online)

The only way to establish quiescence is to use additional code. For example, assuming the number of inserts performed by each rank is unpredictable, we could change the distributed hash table insert loop as follows:

```
// keep track of how many inserts this rank has injected
long n_inserts_injected = 0;
// 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;
  if (should_perform_insert(i)) { // unpredictable condition
```

```
    // insert mapping from key to value in our distributed map.
    // insert has no return because it uses rpc_ff.
    dmap.insert(key, val);
    // increment the local count
    n_inserts_injected++;
  }
  // periodically call progress to allow incoming RPCs to be processed
  if (i % 10 == 0) upcxx::progress();
}

bool done;
do { // Loop while not all insert rpc_ff have completed.
  upcxx::progress(); // process incoming work
  // On each rank, capture the counts of inserts injected and completed
  long local[2] = {n_inserts_injected, dmap.local_size()};
  // Globally count the number of inserts injected and completed by completing
  // an element-wise sum reduction of each of the two counters in the local
  // array, delivering the results in the global array.
  long global[2];
  upcxx::reduce_all(local, global, 2, upcxx::op_fast_add).wait();
  // Test if all inserts have now completed
  UPCXX_ASSERT(global[0] >= global[1]);
  done = (global[0] == global[1]);
} while (!done);
```

(see full example online)

To track completion, each process tracks how many inserts it has injected in the local variable `n_inserts_injected`. The number of inserts that have arrived and completed on each process is tracked implicitly by the local size of the distributed map which is defined in `DistrMap` simply as:

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

Once all processes have finished injecting their inserts, a global sum reduction (`upcxx::reduce_all`) is used to simultaneously compute the total number of inserts injected and completed across all processes. The call to `upcxx::reduce_all` computes an element-wise sum reduction of each of the two counters in the `local` array, storing the results in the `global` array. Reductions are discussed in more detail in the Reduction section. This reduction is repeated until the two values are equal, indicating that all inserts (i.e., `rpc_ff`) have executed.

Note that to safely perform repeated `rpc_ff` quiescence (e.g. in a loop) using the above method requires double buffering of the counters for injected and executed `rpc_ff` – a more complete example of this is provided in the UPC++ Extras package under `examples/rpc_ff_quiescence`.

This counting algorithm is one mechanism to achieve quiescence, which relies on counting the number of one-way messages injected and received and establishing global consensus for when they match. There are many alternative ways to establish quiescence. For example, when the number of messages to be received by each rank is known beforehand or can be efficiently computed, one can deploy a simpler quiescence algorithm that skips the communication step to establish consensus and has each rank simply await the arrival of the expected number of messages.

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 abstraction that encapsulates a supported type and set of

operations. Currently, the supported types include: `float`, `double`, and any signed or unsigned integral type with a 32-bit or 64-bit representation. 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;
  if (should_perform_insert(i)) { // unpredictable condition
    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();
```

(see full example online)

This quiescence algorithm is likely to be slower and less scalable than the example in the Quiescence section, but it's used here to demonstrate the use of remote atomic memory operations. Each rank allocates an `int64_t` counter in the shared space and shares the global pointer to this location with other ranks using a

`dist_object`. This is the location that remote ranks will atomically update, which must be an object of one of the supported scalar types that lives in the shared segment. Next we declare an atomic domain, which includes only the two operations we will actually use:

```
upcxx::atomic_domain<int64_t> ad({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 synchronously wait for each `dist_object::fetch` and `atomic_domain::add` operation to complete to simplify the presentation. However we could easily use future chaining and conjoining or promises to track completion of both operations and achieve full communication/communication overlap for all the operations in the loop; the details are left as an exercise for the reader.

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 examples of completions using promises and RPCs, and a subsequent section demonstrates LPC completions.

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 completion indicates that the source memory resources needed for the operation are no longer in use by UPC++ at the source process, whereas operation and remote completion indicate that the operation is fully complete from the perspective of the initiating process and remote target process, respectively. 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 Cx=/*unspecified*/>
RType upcxx::rput(T const *src, upcxx::global_ptr<T> dest, std::size_t count,
                  Cx &&completions=upcxx::operation_cx::as_future());
```

As shown above, the completion object is constructed as a completion stage combined with a mechanism of completion notification. There are restrictions on which completion notifications 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
```

10.1 Promise Completion

Every asynchronous communication operation has an associated promise object, which is either created explicitly by the user or implicitly by the runtime when a non-blocking operation is invoked requesting the default future-based completion. A promise represents the producer side of an asynchronous 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.

Promises are particularly efficient at keeping track of multiple asynchronous operations, essentially acting as a dependency counter. Here is an example using RMA:

```
upcxx::promise<> p;
upcxx::global_ptr<int> gps[10] = /* . . . */;
for (int i = 0; i < 10; ++i) // register some RMA operations on p:
  upcxx::rput(i, gps[i], upcxx::operation_cx::as_promise(p));
upcxx::future<> f = p.finalize(); // end registration on p, obtain future
f.wait(); // await completion of all RMAs above
```

The first line explicitly constructs a promise, where the empty template arguments indicate it will only track readiness status and not contain/produce a value (an "empty promise"). The default promise constructor initializes its encapsulated dependency counter to 1, placing it in a non-readied state (the promise is ready when the counter reaches zero). Member functions on `promise` allow direct manipulation of the dependency counter, but those are unnecessary for many simple use cases.

Next we issue a series of RMA put operations, overriding the default future-based completion by instead passing `operation_cx::as_promise(p)` to request promise-based completion notification on the supplied promise. Each communication operation has a side-effect of incrementing the dependency counter of the supplied promise at injection time, in a process known as "registering" the communication operation with the promise. When each asynchronous operation later completes, the dependency counter of the same promise will be decremented to provide completion notification. After registering our communication on the promise, we invoke `p.finalize()` which decrements the counter once, matching the counter initialization, and ending the registration stage of the promise. `finalize()` returns the `upcxx::future<>` handle corresponding to our promise, which will be readied once the encapsulated dependency counter reaches zero, indicating that all of the registered asynchronous operations have reached operation completion.

In the following example, we show how promises can be used to track completion of the distributed hash table inserts. 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, const std::string &key, const 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();
  }
  upcxx::future<> fut = prom.finalize(); // finalize the promise
  fut.wait(); // wait for the operations to complete
```

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.

It's instructive to compare this example with the similar example shown in the earlier section on Conjoining Futures. In that prior example, we used future conjoining to dynamically construct a dependency graph of futures. In that graph each asynchronous `find` operation led to asynchronous execution of a local callback to process the resulting value, and completion of that callback was subsequently conjoined into the graph of dependencies, all leading to one future representing completion of the entire set of operations.

In the example above, we use a *single* `promise` object to track all of the outstanding asynchronous dependencies generated by the asynchronous `insert` operations, and the single corresponding future it produces to represent overall completion of the entire set of operations. We could have applied the future conjoining approach to this example, but that would have dynamically constructed many futures (each with a corresponding implicit promise), instead of a single promise object. Because the promise-based approach shown here constructs fewer objects at runtime, it leads to fewer CPU overheads and should generally be favored in situations where it applies; namely, when launching many asynchronous operations and the dependency graph is a trivial aggregation of all the completions, with no ordering constraints or asynchronous callbacks for individually processing results. This general recommendation applies not only to RPC-based communication, but to all forms of asynchrony (e.g. RMA, LPC and even collective communication), all of which support promise-based completion as an alternative to the default future-based completion.

10.2   Remote Completions

Alternatively, the completion object provided to an RMA put can specify a remote procedure call. While the data payload involved can be of any size and the RPC body can be as simple as modifying a variable on the target process, use of `upcxx::rput(remote_cx::as_rpc)` requires that the payload be TriviallySerializable and the destination address be known to the initiator. Satisfying these requirements enables use of zero-copy RMA (via zero-copy RDMA where appropriate). For large payloads, this can yield a performance advantage over alternatives like `upcxx::rpc_ff()` that rely on copying the payload. By querying the value of a variable modified by the callback, RPC completion objects can be used to mimic the coupling of data transfer with synchronization found in message-passing programming models.

As an example, consider a heat transfer simulation in a 3D domain with a 1D decomposition. During each timestep, each process needs to update its interior cells, exchange halos with neighboring ranks, and update boundary cells once the necessary halos are received. RPC completion objects can be used to determine when these halos have arrived. The body of the RPC simply consists of incrementing a counter variable, which is later compared to the number of neighboring processes. As was the case in the last example, `upcxx::progress` (discussed later) must be called to allocate resources to incoming RPCs. The execution of the RPC callback guarantees that the data contained in the `rput` has been delivered. For this example, this means ghost cells from neighboring processes have arrived and can be used for calculation of the process' boundary cells.

Pseudocode for this technique is shown below. A complete compilable code for the heat transfer example is available in the `example/prog-guide` directory of the implementation.

```
int count = 0; // declared as a file-scope global variable
for (n in neighbors) {
    // RMA put halo to remote process
    upcxx::rput(n.outbuf, n.inbuf, n.outbuf.size(),
                remote_cx::as_rpc([](){ count++; }));
}
// await incoming copies
while (count < neighbors.size())
    upcxx::progress();
count = 0; // reset for next timestep
```

11   Progress

Progress is a key concept of the UPC++ execution model that programmers must understand to make most effective use of asynchrony. The UPC++ framework does not spawn any hidden OS threads to perform asynchronous work such as delivering completion notifications, invoking user callbacks, or advancing its internal state. All such work is performed synchronously inside application calls to the library. 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.

11.1 Discharge

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.

Thought exercise:

The red-black example in an earlier section has a barrier marked "optional". Is this barrier required for correctness? Why or why not? Are there any potential negative consequences to its removal?

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 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*. A reference to the default persona of a thread 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 in subsequent sections. When a thread initiates an asynchronous operation, that operation is registered in the internal state managed by the thread's current (top-of-the-stack) persona. A reference to the current persona of a thread can be retrieved using:

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

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. A more robust approach (as we'll see in an upcoming section) 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.

12.1   Master persona

The thread on each process that first initializes 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 that can execute RPCs destined for this process or initiate collective operations (e.g., `upcxx::barrier()`).

A thread can retrieve a reference to the master persona object by calling:

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

It's important to understand the interaction of the master persona with RPC:

> *Remote Procedure Call (RPC) callbacks always execute sequentially and synchronously inside user-level progress for the thread holding the master persona, which is unique to each process. User-level progress only occurs inside a handful of explicit UPC++ library calls (e.g. `upcxx::progress()` and `upcxx::future::wait()`).*

This implies the following useful properties:

1. RPC callbacks never "interrupt" an application thread. They only execute synchronously inside a UPC++ library call (specified with user-level progress) from the thread holding the master persona.

2. RPC callbacks never run concurrently with other RPC callbacks on the same process. This means there can never be a data race between two RPC callbacks running on the same process. This *remains* true even with multiple application threads concurrently invoking UPC++ library calls, because the thread holding the master persona is unique to each process.

3. In a single-threaded application, RPC callbacks are not concurrent with any other code in the same process. This means single-threaded UPC++ applications should never need inter-thread concurrency control mechanisms (std::mutex, std::atomic, etc).

Note that all the properties above apply specifically to RPC (Remote Procedure Call, as injected by `upcxx::rpc()` and `upcxx::rpc_ff()`). This should *not* be confused with LPC (Local Procedure Call, as injected by `upcxx::persona::lpc()` and `upcxx::persona::lpc_ff()` and discussed in an upcoming section). LPCs are not restricted to the master persona, thus LPC callbacks associated with different personas *can* exhibit concurrency in multi-threaded applications.

12.2   Persona identification

Personas objects do not directly provide comparison operators, but they can be compared by address. For example:

```
if (&upcxx::current_persona() == &upcxx::master_persona())
  std::cout << "This thread's current persona is the master persona" << std::endl;
```

However it's important to keep in mind that each thread owns a stack of one or more active personas, all of which are progressed during UPC++ calls with internal- or user-level progress. Also, initiation of collective operations requires the calling thread to hold the master persona, but it need not be the current persona at

the top of the persona stack. The `persona::active_with_caller()` query may be used to search the entire active persona stack of the calling thread for a particular persona. For example:

```
if (upcxx::master_persona().active_with_caller())
  std::cout << "This thread holds the master persona" << std::endl;
```

12.3 Persona management

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`).

12.4 Multithreading and LPC completion

The following toy 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 a 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 LPC attached to the completion object is delivered to `progress_persona` where the callback sets the `done` boolean to true.

```
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 rget
  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) {
            UPCXX_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();
}
```

(see full example online)

12.5   Master Persona and Progress Threads

As described in an earlier section, each process has a master persona which is special in that it is the only persona that can execute RPCs destined for this process or initiate collective operations.

The master 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 (the *primordial thread*) must first release it before other threads can push it. This is done by invoking the `upcxx::liberate_master_persona()` function after `upcxx::init()` and before any other changes to the persona stack on the primordial thread. 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::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, const std::string &key) -> std::string {
        auto elem = lmap->find(key);
        if (elem == lmap->end()) return std::string(); // not found
        else return elem->second; // key found: return value
      },local_map,key);
}
```

(see full example online)

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,

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 completion notification enqueues an LPC to the `progress_persona` which verifies that the received value corresponds to what was expected and decrements the `lpc_count` counter by one. It's noteworthy that although the LPC's were concurrently enqueued to `progress_persona` by several worker threads, the LPC callbacks are all executed (serially) on the progress thread; for this reason no synchronization is necessary when accessing the `lpc_count` variable.

In the last few lines of the example, the primordial thread awaits completion of the progress thread and worker threads by calling `std::thread::join`. When the progress thread exits, the `upcxx::persona_scope` objects it declared are destructed, releasing the `progress_persona` and master persona, and destroying its default persona (which was not used). The master persona is required for all collective operations, including `upcxx::barrier()` and `upcxx::finalize()`, but it was liberated upon exit of the progress thread. Therefore the primordial thread must re-acquire the master persona using a new `upcxx::persona_scope` in order to coordinate an orderly job teardown,

```
// 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;
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;
      });
  // 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](const string &val) {
                UPCXX_ASSERT(val == key);
                lpc_count--;
              });
        }
        // discharge outgoing find operations before thread exit:
        upcxx::discharge();
    });
  }

  // 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();
  }
```

(see full example online)

12.6   Personas and Thread Exit

In the first example of this section, the `submit_thread` waited in a progress loop until the `progress_thread` completed its operations. However this stall was not necessary in the algorithm, because the submit thread was not waiting to receive any callbacks or perform any additional tasks; the stall loop just incurred wasted CPU overhead.

In the second example, we instead allow the worker threads to exit once they've completed injecting their `find` operations. However because each worker thread has launched asynchronous operations (using its default persona) that might remain in-flight, it is critical that we call `upcxx::discharge()` before allowing

the thread to exit. As described in the section on Discharge, this instructs the runtime to ensure that outgoing operations initiated by this thread's personas no longer need further internal progress by this thread to reach completion. Once `discharge` returns, each worker thread may safely terminate, implicitly destroying its default persona.

> *As a general rule, application threads which have launched UPC++ asynchronous operations may not exit, destroying their default persona, until they have:*
>
> 1. *reaped any completion notifications scheduled for that persona (e.g. future readying),* **AND**
> 2. *completed a `upcxx::discharge()` call to ensure any outgoing operations initiated by thread personas no longer require internal progress.*

12.7   Compatibility with Threading Models

The examples presented in this chapter thus far use C++ `std::thread` for simplicity of presentation. However UPC++ is designed to be agnostic to the specific threading model, in order to improve interoperability with a variety of on-node programming models. UPC++ interoperates with other threading mechanisms including OpenMP or POSIX threads, where the same general concepts and techniques apply.

The next section will demonstrate use of UPC++ with OpenMP.

12.8   OpenMP Interoperability

Some care must be exercised when mixing blocking operations provided by two threaded libraries (such as UPC++ and OpenMP) to avoid deadlock. It's important to understand that a thread blocked by one library will generally only advance the internal state of that library. This means if one thread is blocking for an OpenMP condition and another thread is blocking for a UPC++ condition, then they might not be attentive to servicing work from the opposite library, potentially creating a deadlock cycle.

A simple example involves a thread waiting at an OpenMP barrier (which is implicit upon the closing brace of an OpenMP parallel region) while another thread from that same region sends it a UPC++ LPC. The recipient thread is inattentive to UPC++ progress, meaning the sending thread will never have its LPC completed no matter how many times `upcxx::progress` is called by the initiating thread. As such, it is unable to join the other thread at the barrier, leading to deadlock.

A similar deadlock scenario appears below, where, where one thread stalls in `upcxx::future::wait()` awaiting completion of a round-trip UPC++ RPC, but the thread holding the master persona at the target process has stalled in an OpenMP barrier, preventing the RPC from executing.

```cpp
upcxx::intrank_t me = upcxx::rank_me();
upcxx::intrank_t n = upcxx::rank_n();

#pragma omp parallel num_threads(2)
{
  // Assume thread 0 has the master persona...

  if(omp_get_thread_num() == 1) {
    // Bounce an rpc off buddy rank and wait for it.
    upcxx::rpc((me^1) % n,[=](){}).wait();
  }
  // Here comes the implicit OpenMP barrier at the end of the parallel region.
  // While in the OpenMP barrier, the master persona can't respond to RPCs. So
  // we can't reply to our buddy's message and therefore delay the buddy's
  // sending thread from making it to the OpenMP barrier. And if our buddy is
  // in the same situation with respect to our message then we deadlock.
}
```

Two solutions will be provided in this section for avoiding this type of deadlock. The first involves separating the thread with the master persona from other threads so it can explicitly service incoming work. Inside the `omp parallel` region, the other threads will perform the same work as before, while the master persona thread will execute a `upcxx::progress` loop. The satisfaction of that loop condition should indicate that all local worker threads have had their RPCs completed, which implies that the master persona can safely proceed to the implicit OpenMP barrier and be guaranteed that thread barrier will complete immediately. An example is shown below, which uses an atomic integer to track whether all non-master threads have completed the RPC they issued. Once the counter indicates this, the master thread joins the other threads at the OpenMP barrier.

```cpp
const int me = upcxx::rank_me();
const int n = upcxx::rank_n();
const int buddy = (me^1)%n;
const int tn = thread_count; // threads per process
std::atomic<int> done(1); // master thread doesn't do worker loop

#pragma omp parallel num_threads(tn)
  {
    UPCXX_ASSERT(tn == omp_get_num_threads());
    // OpenMP guarantees master thread has rank 0
    if (omp_get_thread_num() == 0) {
      UPCXX_ASSERT(upcxx::master_persona().active_with_caller());
      do {
        upcxx::progress();
      } while(done.load(std::memory_order_relaxed) != tn);
    } else { // worker threads send RPCs
      upcxx::future<> fut_all = upcxx::make_future();
      for (int i=0; i<10; i++) { // RPC with buddy rank
        upcxx::future<> fut = upcxx::rpc(buddy,[](int tid, int rank){
          std::cout << "RPC from thread " << tid << " of rank "
          << rank << std::endl;
        },omp_get_thread_num(),me);
        fut_all = upcxx::when_all(fut_all,fut);
      }
      fut_all.wait(); // wait for thread quiescence
      done++; // worker threads can sleep at OpenMP barrier
    }
  } // <-- this closing brace implies an OpenMP thread barrier

upcxx::barrier(); // wait for other ranks to finish incoming work
```

(see full example online)

As mentioned previously, UPC++ futures and promises are persona-specific, even with `UPCXX_THREADMODE=par`. As such, when using them with OpenMP threads it's important to make sure a future or promise is not shared between threads. This example is safe because by default variables declared inside an OpenMP parallel region are thread-private.

It should be noted that this method only works when just the master thread is needed to execute incoming code. When all threads are needed, then they all must enter a loop that calls `upcxx::progress()` until local UPC++ quiescence is achieved. An example of this approach appears below. The body of the OpenMP `for` loop has an `rput` that puts the process' rank in global memory. The completion object is an LPC that executes on an arbitrary local thread, and executes an `rget` which simply validates the correct value was put and increments an atomic counter to check that all remote operations are done. However, the `rget` is in an LPC that runs on a (potentially) different persona than the `rput`. The `while` loop on the final two lines before the closing brace ensures that none of the threads holding the personas are put to sleep by the

OpenMP runtime until all callbacks have finished. Until then, progress is called to service them.

```cpp
const int n = upcxx::rank_n();
const int me = upcxx::rank_me();
const int tn = thread_count;  // threads per process

vector<upcxx::global_ptr<int>> ptrs = setup_pointers(n);
std::vector<upcxx::persona*> workers(tn);

std::atomic<int> done_count(0);

#pragma omp parallel num_threads(tn)
{
  // all threads publish their default persona as worker
  int tid = omp_get_thread_num();
  workers[tid] = &upcxx::default_persona();
  #pragma omp barrier

  // launch one rput to each rank
  #pragma omp for
  for(int i=0; i < n; i++) {
    upcxx::rput(&me, ptrs[(me + i)%n] + me, 1,
      // rput is resolved in continuation on another thread
      upcxx::operation_cx::as_lpc(*workers[(tid + i) % tn], [&,i]() {
        // fetch the value just put
        upcxx::rget(ptrs[(me + i)%n] + me).then([&](int got) {
          UPCXX_ASSERT_ALWAYS(got == me);
          done_count++;
        });
      })
    );
  }

  // each thread drains progress until all work is quiesced
  while(done_count.load(std::memory_order_relaxed) != n)
    upcxx::progress();
} // <-- this closing brace implies an OpenMP thread barrier

upcxx::barrier();
```

(see full example online)

Lastly, while the examples in this section used OpenMP constructs to launch asynchronous operations, OpenMP can also be used within the operations themselves. However, it should be noted that in a UPC++ context the same limitations on parallelism apply as in standalone OpenMP code. This means for instance that if a callback containing `#pragma omp parallel` is dynamically nested within a code block that has its own parallel construct, then there won't be any additional parallelism to utilize since all threads are in use. If a developer wants to ensure that callbacks have access to more threads than the code block from which they are run, he can use the `omp_set_max_active_levels()` routine.

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.

13.1 `team::split`

New `upcxx::team` objects can be created by collectively 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);
```

(see full example online)

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 `team::split()` function is a very powerful and convenient means for subdividing teams into smaller teams, and/or creating a new team that renumbers process ranks in an existing team. However this function semantically requires collective communication across the parent team to establish new team boundaries, and thus can entail non-trivial cost when the parent team is large.

13.2 `team::create`

In cases where processes can cheaply/locally compute the membership of a new team they wish to construct, the `team::create()` member function enables team construction with lower communication overheads than `team::split()`. Here is an example that further partitions the `new_team` created in the prior example, assigning processes into pair-wise teams:

```
upcxx::intrank_t group = new_team.rank_me() / 2; // rounds-down
upcxx::intrank_t left  = group * 2;
upcxx::intrank_t right = left + 1;
std::vector<upcxx::intrank_t> members({left});
if (right != new_team.rank_n()) // right member exists
  members.push_back(right);
upcxx::team sub_team = new_team.create(members);
```

(see full example online)

As with `team::split()`, the `team::create()` function is collective over a parent team, in this case `new_team`. Each caller passes an ordered sequence of ranks in that parent team, enumerating the participants of the new sub team it will join. Here we see each process uses `new_team.rank_me()` to query its rank in the parent team and uses that to construct a `std::vector` of rank indexes selecting members for the team it will construct. `team::create()` accepts any ordered C++ Container, or alternatively a pair of InputIterators delimiting the rank sequence. Either way, the sequence passed is required to match across all callers joining the same team, and (as with `team::split()`) each process joins (at most) one team per call.

13.3 Team Accessors

As previously mentioned, the team member functions `rank_me()` and `rank_n()` can be used to respectively retrieve the rank of the calling process in that team and the number of ranks in the team. There is also an `id()` function that returns a trivially copyable `team_id` object representing a universal name identifying 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 the first example of this section 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();
  UPCXX_ASSERT(world_rank == expected_world_rank);

  upcxx::intrank_t other_local_rank = new_team.from_world(world_rank);
  UPCXX_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);
  UPCXX_ASSERT(non_member_rank == -1);

  new_team.destroy(); // collectively release the sub-team
```
(see full example online)

13.4   Local Team

In addition to the `upcxx::world()` team, another special team is available that represents all the processes sharing the same physical shared-memory node. This team is obtained by calling the `upcxx::local_team()` function, and has the following very useful property:

> All members of `local_team()` with a `upcxx::global_ptr<T>` `gptr` *referencing shared memory allocated by* **any** *member of that team are guaranteed to see* `gptr.is_local() == true`, *and* `gptr.local()` *will return a valid* `T*` *(raw C++ pointer) to the memory.*

This is particularly important if one wants to optimize for *shared-memory bypass*. For example, it means that one can apply the techniques described in the section Downcasting global pointers to obtain direct C++ pointers to shared objects owned by other processes who are members of `local_team()`. This works because at startup the UPC++ runtime ensures that all processes in the `local_team()` automatically map each other's shared host memory segments into virtual memory, allowing for direct load/store access to that memory by the CPU.

In the following example, one process per physical node loads data from a file into shared memory, then it allows access to such data by broadcasting a global pointer to it. By doing this, we avoid multiple processes loading the same data, and we improve memory scalability by a factor of `local_team().rank_n()`.

```
  size_t n = 0;
  upcxx::global_ptr<double> data;

  if (!upcxx::local_team().rank_me()) { // I'm the leader process in this node
    // Open the file in binary mode
    std::ifstream input_file(filename, std::ios::binary);

    // Ensure any I/O errors will throw an exception
    input_file.exceptions(std::ifstream::failbit);

    // How many elements am I supposed to read?
    input_file.read(reinterpret_cast<char*>(&n), sizeof(n));

    data = upcxx::new_array<double>(n); // Allocate space in shared memory

    // Read the entire array of doubles from the file
    input_file.read(reinterpret_cast<char*>(data.local()), sizeof(double)*n);
  }

  // Leader broadcasts data to other processes in the local team (implicit barrier)
  std::tie(n, data) =
    broadcast(std::make_tuple(n, data), 0, upcxx::local_team()).wait();

  double *ldata = data.local(); // Downcast global pointer

  process_data(n, ldata); // Access shared data using local ptr
```
(see full example online)

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.

14.1 Barrier

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

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

template<typename Cx=/*unspecified*/>
RType upcxx::barrier_async(upcxx::team &team = upcxx::world(),
                           Cx &&completions=upcxx::operation_cx::as_future());
```

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.

14.1.1 Interaction of collectives and operation completion

It is important to note that although collectives must be issued in the same order by all participants, asynchronous operations (collective or otherwise) are not guaranteed to complete in any particular order relative to other in-flight operations. In particular:

> *Neither `barrier` nor `barrier_async` act as a "flush" of outstanding asynchronous operations. UPC++ does not provide* ***any*** *calls that implicitly "fence" or "flush" outstanding asynchronous operations; all operations are synchronized explicitly via completions.*

In other words, issuing and completing a barrier does NOT guarantee completion of asynchronous operations issued before the barrier and not yet explicitly synchronized.

This property may seem counter-intuitive, but it enables the communication layer to maximize utilization of the network fabric, which is often fundamentally unordered at the lowest levels. As discussed in Asynchronous Computation, relaxed ordering guarantees for asynchronous operations enable more efficient implementations, but the programmer must not assume any ordering of asynchronous operation completion that is not enforced by explicit synchronization. As discussed in Quiescence, issuing an asynchronous operation and failing to later synchronize its completion is an error. A corollary of these properties:

> *Discarding a `upcxx::future` returned from a UPC++ call is almost always a bug.*

In fact, the implementation uses compiler annotations (where supported) to encourage a compiler warning for this type of potential defect.

14.2 Broadcast

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 Cx=/*unspecified*/>
RType upcxx::broadcast(T value, upcxx::intrank_t root,
                       upcxx::team &team = upcxx::world(),
                       Cx &&completions=upcxx::operation_cx::as_future());


template <typename T, typename Cx=/*unspecified*/>
RType upcxx::broadcast(T *buffer, std::size_t count, upcxx::intrank_t root,
                       upcxx::team &team = upcxx::world(),
                       Cx &&completions=upcxx::operation_cx::as_future());
```

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<>`. All participants in a broadcast collective must "agree" on which team member is the source of the broadcast, and in particular all callers must pass the same value for the `root` argument. This is referred to as a *single-valued* constraint in a collective operation; in the second `broadcast` variant the `count` argument is also required to be single-valued.

These functions can be used in the following way:

```
std::vector<int> buffer;
size_t elem_count{};
if (upcxx::rank_me() == 0) {
  elem_count = 42; // initially only rank 0 knows the element count
}
// launch a scalar broadcast of element count from rank 0
upcxx::future<size_t> fut = upcxx::broadcast(elem_count, 0);

// do some overlapped work like preparing buffer for next broadcast
```

```
if (upcxx::rank_me() == 0) {
  buffer.reserve(elem_count);
  for (size_t i = 0; i < elem_count; i++)
    buffer[i] = (int)i;
}

elem_count = fut.wait(); // complete first broadcast
buffer.reserve(elem_count); // non-zero ranks allocate vector space

// launch a bulk broadcast of element data from rank 0
upcxx::future<> fut_bulk = upcxx::broadcast( buffer.data(), elem_count, 0);

// wait until the second broadcast is complete
fut_bulk.wait();

// consume elements
for (size_t i = 0; i < buffer.size(); i++)
  UPCXX_ASSERT_ALWAYS(buffer[i] == (int)i);
```

(see full example online)

14.3   Reduction

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 , Cx=/*unspecified*/>
RType upcxx::reduce_one(T value, BinaryOp &&op, upcxx::intrank_t root,
          upcxx::team &team = upcxx::world(),
          Cx &&completions=upcxx::operation_cx::as_future());

template <typename T, typename BinaryOp, Cx=/*unspecified*/>
RType upcxx::reduce_all(T value, BinaryOp &&op,
          upcxx::team &team = upcxx::world(),
          Cx &&completions=upcxx::operation_cx::as_future());

template <typename T, typename BinaryOp, Cx=/*unspecified*/>
RType upcxx::reduce_one(const T *src, T *dst, size_t count, BinaryOp &&op,
          upcxx::intrank_t root, upcxx::team &team = upcxx::world(),
          Cx &&completions=upcxx::operation_cx::as_future());

template <typename T, typename BinaryOp, Cx=/*unspecified*/>
RType upcxx::reduce_all(const T *src, T *dst, size_t count, BinaryOp &&op,
          upcxx::team &team = upcxx::world(),
          Cx &&completions=upcxx::operation_cx::as_future());
```

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 a "multi-field" element-wise reduction on contiguous buffers of `count` objects of type `T` pointed by `src` (where `count` must be single-valued). 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 (which must be single-valued), and the result is undefined on other processes. For example:

```
int sum_to_P0 = upcxx::reduce_one(upcxx::rank_me()*42, upcxx::op_fast_add, 0).wait();
```

On the other hand, when using the `upcxx::reduce_all` functions, a reduction result will be provided to all processes belonging to `team`. For example:

```
int sum_to_all = upcxx::reduce_all(upcxx::rank_me()*42, upcxx::op_fast_add).wait();
```

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.

In the following example, each process finds its local minimum value (of type `std::complex<double>`), and then it calls `upcxx::reduce_one` providing a *lambda* as the `op` argument and 0 as the root process,

```cpp
    // Find local minimum
    Complex lmin = *std::min_element(arr, arr + N, [](const Complex& a, const Complex& b) {
        return std::norm(a) < std::norm(b);
        });

    // Find global minimum and its location (MINLOC)
    auto gmin = upcxx::reduce_one(std::make_pair(lmin, upcxx::rank_me()),
        [](const std::pair<Complex, int>& a, const std::pair<Complex, int>& b) {
            auto norma = std::norm(a.first), normb = std::norm(b.first);
            if (norma == normb && b.second < a.second) return b;
            else return (norma <= normb ? a : b);
        }, 0).wait();
```

(see full example online)

Notice that the comparison involves both components of the pair. That is because reduction routines require `op` to be associative and commutative. In the above example, if two or more processes provide the same `Complex` value, we use their rank to break the tie and guarantee the reproducibility of the operation. In general, user-provided reduction operators over a class type that return a provided input value (e.g., min or max) will need to consult all the non-static fields of the operands when some fields happen to be equal, in order to ensure commutativity of the operation.

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 specialized forms of these RMA functions for moving non-contiguous 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 even more specialized: there is just one base address specified for each of the source and destination regions, together with stride vectors describing a dense multi-dimensional array transfer.

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

constexpr ptrdiff_t elem_sz = (ptrdiff_t)sizeof(float);

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

Figure 4: *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, {{elem_sz, sdim[0]*elem_sz, sdim[0]*sdim[1]*elem_sz}},
      dst_base, {{elem_sz, ddim[0]*elem_sz, ddim[0]*ddim[1]*elem_sz}},
      {{4, 3, 2}});
}
```
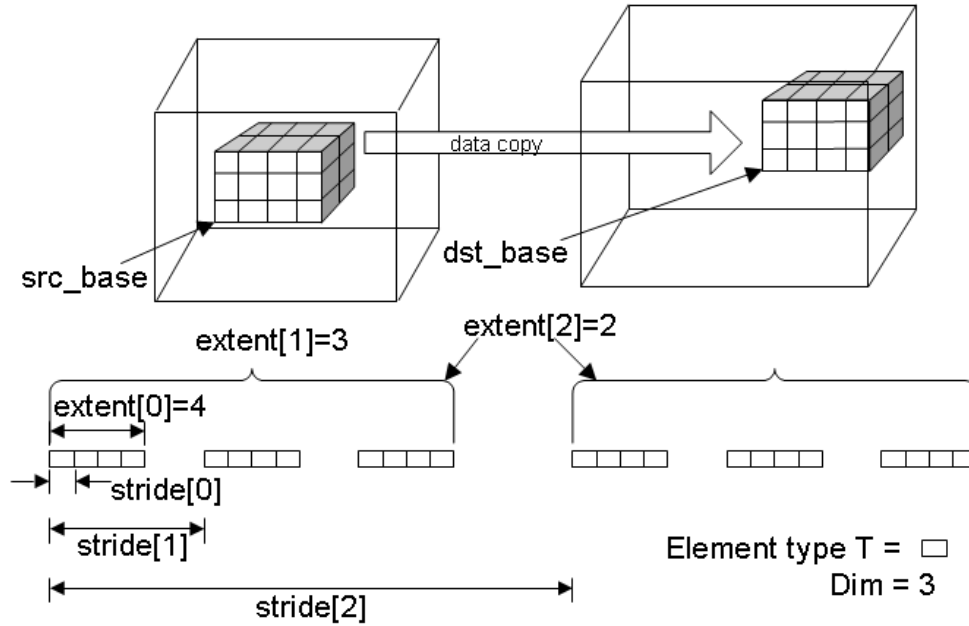
(see full example online)

The strided example code snippet above corresponds to the translation data motion shown in Figure 4. By altering the arguments to the `_strided` functions a user can implement various transpose and reflection operations in an N-dimensional space within their non-contiguous RMA operation. Another common use case is performing RMA on a non-contiguous boundary plane of a dense multi-dimensional array, for example as part of a rectangular halo exchange in a simulation with a multi-dimensional domain decomposition. 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();
```

(see full example online)

The user here is taking subsections of a source array with `particle_t` element type pointed to by `srcP` and copying them to the location pointed to by `destP`. This example also shows how data can be shuffled in a non-contiguous RMA operation; the user is responsible for ensuring their source and destination areas specify equivalent amounts of data. As with all RMA operations, all variants of rput and rget assume the element type is TriviallySerializable (byte-copyable).

43

16   Serialization

RPC's transmit their arguments across process boundaries using the mechanism of *serialization*, which is type-specific logic responsible for encoding potentially rich C++ objects to and from raw byte streams. UPC++ provides built-in support for serializing trivially copyable types (e.g. primitive types, and C-like structs or other classes which report as `std::is_trivially_copyable`). It also supports serialization for most STL containers (e.g. `std::vector`, `std::list`) when the elements they contain are trivially copyable (or otherwise Serializable). However, in many cases it may be necessary to add application-specific serialization for application-specific objects. Common motivations for this include:

1. Sending objects that are not trivially copyable through RPC (e.g. some application-specific object that uses custom data structures).
2. Transforming trivially copyable objects into a meaningful representation on the target process of the RPC. For example, while raw C++ pointer members of classes are trivially copyable, the addresses they contain are usually not meaningful on a remote process.
3. Limiting the attributes of an object that are serialized and transmitted. This may be useful if a large object is being transmitted with an RPC, but only a small subset of its members are used in the remote logic.
4. Reducing overheads from serializing and deserializing container objects. Deserializing byte streams into C++ objects adds computational overheads that may not be useful if the application logic could be implemented to directly consume elements from the byte stream representation (e.g., without constructing a new container around those elements at the target process).

16.1   Serialization Concepts

UPC++ defines the concepts *TriviallySerializable* and *Serializable* that describe what form of serialization a C++ type supports. Figure 5 helps summarize the relationship of these concepts.
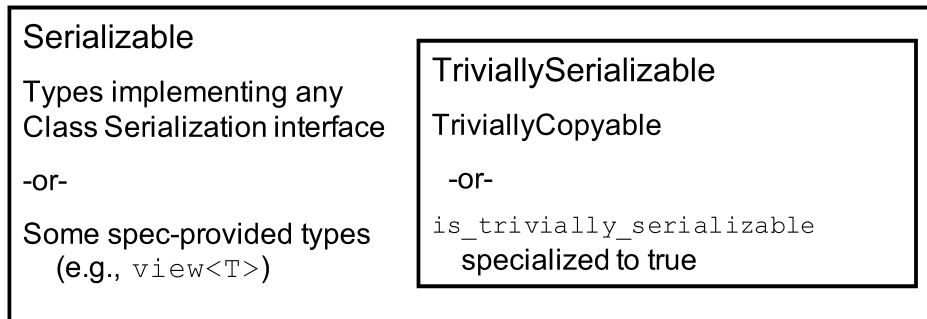
Figure 5: *Serialization Concepts.*

A type `T` is TriviallySerializable if it is semantically valid to copy an object by copying its underlying bytes, and UPC++ serializes such types by making a byte copy. A type `T` is considered to be TriviallySerializable if either of the following holds:

- `T` is TriviallyCopyable (i.e. `std::is_trivially_copyable<T>::value` is `true`), and does not implement any of the class serialization interfaces described in the following sections.
- `upcxx::is_trivially_serializable<T>` is specialized to provide a member constant `value` that is `true`.

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

Certain UPC++-provided types such as `upcxx::global_ptr<T>` and `upcxx::team_id` are also defined as TriviallySerializable.

A type `T` is Serializable (the more general concept) if one of the following holds:

- `T` is TriviallySerializable
- `T` is of class type and implements one of the class serialization interfaces described in the following sections.
- `T` is explicitly defined as Serializable by the UPC++ specification. This includes STL containers (`std::vector`, `std::tuple`, `std::array`, `std::map` etc.) where the elements are Serializable, and certain other types including `std::string` and `upcxx::view<T>`.

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

```
namespace upcxx {
  template<>
  struct is_trivially_serializable<::T> : ::std::true_type {};
}
```

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

Many UPC++ communication operations (notably RMA and collectives) assert that the objects to be transferred are of TriviallySerializable type. This is to ensure these operations are amenable to hardware acceleration, such as the use of RDMA network hardware to accelerate `rput` and `rget` operations. UPC++ RPC operations are inherently two-sided (they semantically require CPU interaction on both sides), and thus arguments to RPC are only required to be Serializable (not restricted to TriviallySerializable).

Note that serializability of a type `T` does not imply that objects of type `T` are meaningful on another process. In particular, C++ pointer-to-object and pointer-to-function types are TriviallySerializable, but it is generally invalid to dereference a local pointer that originated from another process. More generally, objects that represent local process resources (e.g., file descriptors) are usually not meaningful on other processes, whether their types are Serializable or not.

The following sections discuss three ways in which UPC++ programmers can implement application-specific serialization logic: field/value-based serialization, custom serialization, and view-based serialization. Each method offers different tradeoffs on productivity, flexibility, and performance.

16.2 Field- and Value-Based Serialization

The simplest and easiest-to-use mechanisms for defining serialization of application-specific class types in UPC++ are the `UPCXX_SERIALIZED_FIELDS` and `UPCXX_SERIALIZED_VALUES` macros, which specify a set of member fields or values to comprise the serialized representation of a given type.

In `UPCXX_SERIALIZED_FIELDS` the programmer passes a list of non-static member field names in a class, and the UPC++ library automatically generates the needed serialization logic to transmit those fields (and only those fields) whenever an instance of that class is sent as part of an RPC. These fields themselves must be Serializable. For `UPCXX_SERIALIZED_FIELDS` to be applied to a class, that class must also have a default constructor.

In `UPCXX_SERIALIZED_VALUES` the programmer passes a list of expressions to be evaluated every time an object of the given class type is serialized. The results of these expressions are then transmitted, and must themselves be Serializable. For a class to use `UPCXX_SERIALIZED_VALUES` it must also declare a constructor that accepts the values resulting from the expressions passed to `UPCXX_SERIALIZED_VALUES`.

The example below illustrates using `UPCXX_SERIALIZED_FIELDS` to serialize a large object containing a large nested array. While this object is Serializable without `UPCXX_SERIALIZED_FIELDS`, using `UPCXX_SERIALIZED_FIELDS` allows us to significantly reduce the number of bytes transmitted each time an instance of this class is serialized (assuming only the `partial_sum_reduction` field is needed in the remote logic, and not `values`).

```
class dist_reduction {
    public:
        // The values to perform a sum reduction across
        double values[N];
        // Used to store a local sum reduction result on each rank
        double partial_sum_reduction;

        // Default constructor used by UPC++ deserialization
        dist_reduction() {
            for (int i = 0; i < N; i++) { values[i] = 1.; }
        }

        void calculate_partial_sum_reduction() {
            partial_sum_reduction = 0.0;
            for (int i = 0; i < N; i++) { partial_sum_reduction += values[i]; }
        }

        UPCXX_SERIALIZED_FIELDS(partial_sum_reduction)
};
```
(see full example online)

The following example implements similar semantics to the previous example code, but using `UPCXX_SERIALIZED_VALUES`. Note the addition of a constructor that accepts the single `double` value that comprises the serialized representation. This constructor is invoked at the target process to instantiate the deserialized object.

```
class dist_reduction {
    public:
        // The values to perform a sum reduction across
        double values[N];
        // Used to store a local sum reduction result on each rank
        double partial_sum_reduction;

        // Constructor used by UPC++ deserialization
        dist_reduction(double _partial_sum_reduction) {
            partial_sum_reduction = _partial_sum_reduction;
        }

        dist_reduction() {
            for (int i = 0; i < N; i++) { values[i] = 1.; }
        }

        double calculate_partial_sum_reduction() const {
            double partial_sum_reduction = 0.0;
            for (int i = 0; i < N; i++) { partial_sum_reduction += values[i]; }
            return partial_sum_reduction;
        }

        UPCXX_SERIALIZED_VALUES(calculate_partial_sum_reduction())
};
```
(see full example online)

16.3   Choosing a Serialization Mechanism

The decision tree below can help users determine how to implement serialization for a class of type `T`. It is not exhaustive of all possible use cases, but rather is designed to provide general guidance. Fields `F1,...,Fn` are defined as the non-static members of `T` that encompass all the object state required to represent a serialized `T`.

- Are ALL non-static fields of `T` TriviallyCopyable?

  YES:

  – Is ANY non-static field of `T` meaningful only on the local process (such as a raw C++ pointer, C++ reference, or other "process-local resource")?

    YES: **goto** <NON-TRIVIAL-SERIALIZATION>

    NO:

      * Is `T` a TriviallyCopyable class?

        YES: **Result**: Use trivial serialization, `T` should already be TriviallySerializable

        NO:

          · Does `T` have any non-trivial constructors, assignment operators or destructors that need to perform actions during (de)serialization?

            YES: **goto** <NON-TRIVIAL-SERIALIZATION>

            NO: **Result**: Force trivial serialization by specializing `upcxx::is_trivially_serializable<T>` (see Serialization Concepts)

  NO: **goto** <NON-TRIVIAL-SERIALIZATION>

NON-TRIVIAL-SERIALIZATION:

- Do you have access to change the implementation of `T`?

  YES:

  – Is any field `F1,...,Fn` a raw C++ pointer, a "process-local resource", or a `std` container around these?

    YES:

      * Can you write an expression to convert each field needed by serialization into a meaningfully Serializable object?

        YES: **Result**: `UPCXX_SERIALIZED_VALUES(expr(F1),...,expr(Fn))`

        NO: **Result**: Use Custom Serialization by defining `T::upcxx_serialization`

    NO:

      * Are all of the fields `F1,...,Fn` Serializable and Destructible?

        YES:

          · Is `T` DefaultConstructible AND none of the fields `F1,...,Fn` are references, const-qualified, or `std` container around these?

            YES: **Result**: `UPCXX_SERIALIZED_FIELDS(F1,...,Fn)`

            NO: **Result**: `UPCXX_SERIALIZED_VALUES(F1,...,Fn)`

        NO: **Result**: Implement Serialization for the relevant field types that need it and restart from the top of this decision tree.

  NO: **Result**: Use Custom Serialization by specializing `upcxx::serialization<T>`

16.4   Custom Serialization

While `UPCXX_SERIALIZED_FIELDS` and `UPCXX_SERIALIZED_VALUES` are powerful in their simplicity, there are many use cases that require more complex serialization logic than can be expressed in a simple expression. One common example is deep copying pointers in member fields. While the pointer itself is trivially copyable, it is not useful on a remote process. Usually, the desired behavior is that the object referenced by the pointer is itself serialized into the same byte stream as the object pointing to it so that they can both be reconstructed on the remote rank. However in other cases, we may want such a pointer to be cleared or set to reference a pre-existing object at the target.

For this and other complex logic, UPC++ offers custom serialization. In custom serialization, the programmer is responsible for manually serializing and deserializing a C++ object to/from a byte stream using `Writer` and `Reader` objects to push and pull values into and out of that stream. The programmer implements (1) a `serialize` method that accepts a `Writer` and the object to be serialized., and (2) a `deserialize` method that accepts a `Reader` and a pointer to a block of memory the same size as the type being deserialized.

There are two ways in which these `serialize` and `deserialize` methods can be declared. First, they can be declared as member methods in a public, nested member type named `upcxx_serialization` of the class to be serialized/deserialized. Second, they can be declared in a specialization of `upcxx::serialization<T>` where `T` is the type to be serialized/deserialized, with public `serialize` and `deserialize` methods. These two mechanisms are respectively intended to serve the separate use cases of (1) serialization implemented by the class author, and (2) serialization added to a pre-existing class.

The example below illustrates custom serialization using a public, nested `upcxx_serialization` class for an application-defined `vertex` class which stores a list of neighbor vertices. The `serialize` method writes the ID of the vertex, the number of neighbors it has, and then the ID of each of those neighbors. The `deserialize` method then extracts this information on the remote rank and uses it to reconstruct an instance of `vertex`.

```
class vertex {
      int id;
      std::vector<vertex *> neighbors;
   public:
      vertex(int _id) : id(_id) { }
      int get_id() const { return id; }
      bool has_edge(int other) const;
      void add_neighbor(int neighbor_id);

      // An example of using a member struct upcxx_serialization to implement
      // custom serialization for the vertex class:
      struct upcxx_serialization {
         template<typename Writer>
         static void serialize (Writer& writer, vertex const & object) {
            writer.write(object.get_id());
            writer.write(object.neighbors.size());
            for (vertex *neighbor : object.neighbors) {
               writer.write(neighbor->get_id());
            }
         }
```

```
            template<typename Reader, typename Storage>
            static vertex* deserialize(Reader& reader, Storage storage) {
                int id = reader.template read<int>();
                size_t n_neighbors = reader.template read<size_t>();

                vertex *v = storage.construct(id);
                for (size_t n = 0; n < n_neighbors; n++) {
                    v->add_neighbor(reader.template read<int>());
                }
                return v;
            }
        };
};
```

Note that custom serialization can also be applied to classes that are not defined by the application, which instead might be defined by a library the application is using. A specialization of `upcxx::serialization<T>` can be created to (de)serialize the accessible state of `T`, where `T` is the type to be serialized/deserialized, regardless of where `T` is defined. A more complete version of the above example is available in the `examples/serialization` directory of the implementation repository, which also demonstrates this technique.

During deserialization there are multiple routines available for reading data from the byte stream. The best one to use depends on the amount of data being read, as well as the destination they are being read into. The decision tree helps to illustrate when to use which routine, using `U` as an alias for `deserialized_type_t<T>`.

Implementing Custom Deserialization

- Is there a sequence of objects of the same type being read from the byte stream?

    YES:

    – Does the memory they're being read into already have existing objects?

        YES: `U *ptr = read_sequence_overwrite<T>(array, num_items)`

        NO: `U *ptr = read_sequence_into<T>(storage, num_items)`

    NO:

    – Is object being read into a variable or heap storage?

        VARIABLE:

        * Does the variable contain an object that needs to be destructed before the object in the byte stream is read in?

            YES: `U *ptr = read_overwrite<T>(var)`

            NO: `U var = read<T>()`

        STORAGE:

        * Does the storage point to an object that needs to be destructed before the object in the byte stream is read in?

            YES: `U *ptr = read_overwrite<T>(*storage)`

            NO: `U *ptr = read_into<T>(storage)`

16.5 Recursive Serialization

It is often the case that an application-defined class requiring custom serialization itself contains member fields which are also of application-defined types. While a programmer could implement the serialization/deserialization of the wrapping class by extracting data from those member fields and then re-initializing them during deserialization, this logic would have to be duplicated in every place where instances of those classes were also being serialized/deserialized. Instead, UPC++ supports recursive serialization wherein the serialization logic for one class may implicitly call the serialization logic for another.

The example below illustrates how this looks in the case of `UPCXX_SERIALIZED_FIELDS`. `custom_class_1` is Serializable because it uses `UPCXX_SERIALIZED_FIELDS` to serialize its only member field (`msg`) which is itself of a Serializable type (`std::string`). `custom_class_2` then uses `UPCXX_SERIALIZED_FIELDS` to serialize its member field `std::vector<custom_class_1> msgs`. `msgs` is Serializable thanks to UPC++'s built-in support for `std::vector`, and because `custom_class_1` has already used `UPCXX_SERIALIZED_FIELDS` to declare how it should be serialized.

This enables a separation of concerns, wherein programmers do not need to recursively and manually serialize deep object hierarchies themselves so long as the nested objects are themselves Serializable.

```
class custom_class_1 {
    public:
        std::string msg;
        custom_class_1() { }
        custom_class_1(const std::string &_msg) : msg(_msg) { }

        UPCXX_SERIALIZED_FIELDS(msg)
};

class custom_class_2 {
    public:
        std::vector<custom_class_1> msgs;
        custom_class_2() { }
        void add_msg(const std::string &m) { msgs.push_back(custom_class_1(m)); }

        UPCXX_SERIALIZED_FIELDS(msgs)
};
```

(see full example online)

UPC++ also supports hierarchical serialization of a base class from within a derived class. Both `UPCXX_SERIALIZED_FIELDS` and `UPCXX_SERIALIZED_VALUES` accept `UPCXX_SERIALIZED_BASE(B)` as an argument, where B is the type of the base class (not the member within the derived class). Derived classes can also use custom serialization as long as the base class has a serializable type. For details, see the *UPC++ Specification*.

16.6 View-Based Serialization

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 during deserialization:

1. UPC++ would construct and populate a `vector` by visiting each `T` element in the network buffer and copying it into the `vector` container.

50

2. UPC++ would invoke the RPC callback function, passing 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 considerable unnecessary overhead that might be problematic in performance-critical communication. The remedy to eliminating the overheads associated with steps 1 and 4 is to allow the application 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.

16.6.1 Reducing Overheads with Views

The following example demonstrates how a user could easily employ views to implement a remote vector accumulate, i.e., adding the values contained in a local buffer to the values in a remote shared array. Views enable the transmission of the local 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, adding each element to the corresponding element in a shared array on the target rank.

```
upcxx::future<> add_accumulate(upcxx::global_ptr<double> remote_dst,
                               double *buf, std::size_t buf_len) {
  return upcxx::rpc(remote_dst.where(),
    [](const upcxx::global_ptr<double>& dst, const upcxx::view<double>& buf_in_rpc) {
      // Traverse `buf_in_rpc` like a container, adding each element to the
      // corresponding element in dst. Views fulfill most of the container
      // contract: begin, end, size, and if the element type is trivial, even operator[].
      double *local_dst = dst.local();
      std::size_t index = 0;
      for(double x : buf_in_rpc) {
        local_dst[index++] += x;
      }
    },
    remote_dst, upcxx::make_view(buf, buf + buf_len));
}
```

(see full example online)

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 views, illustrating the performance advantages that views enable.

```
// Hash a key to its owning rank.
```

```cpp
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;
  // 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();
}

// 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;
  // 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>> const &histo_view) {
        // Pain point from `send_histo1_byval`: Eliminated.

        // Traverse key-values directly from 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();
}
```

(see full example online)

16.6.2 Subset Serialization with Views

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;
  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>> const &histo_view) {
        // Traverse key-values directly from 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();
}
```

16.6.3   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 = /*...*/;
upcxx::view<T> tmp = upcxx::make_view(vec); // Type error: mismatched Iter types
auto tmp = upcxx::make_view(vec); // OK: deduced upcxx::view<T,std::vector<T>::const_iterator>
```

16.6.4   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 callback function is executing. In fact, UPC++ will actually extend this lifetime until the future (if any) returned by the RPC callback is ready. This gives the application a convenient means to dispatch the processing of the incoming view to another concurrent execution agent (e.g. a thread), thereby returning from the RPC callback nearly immediately and allowing the UPC++ runtime to resume servicing additional user 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.

```
double my_matrix[1000][1000] = {/*0...*/}; // Rank's local matrix.
constexpr int worker_n = 8; // Number of worker threads/personas.

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

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

```cpp
upcxx::future<> update_remote_matrix(upcxx::intrank_t rank,
                                     element const *elts, int elt_n) {
  return upcxx::rpc(rank,
    [](upcxx::view<element> const &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 const &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)
  );
}
```

(see full example online)

These lifetime extension semantics for RPC callback arguments actually apply not only to `upcxx::view` (as demonstrated above), but also to all deserialized arguments that an RPC callback accepts by const reference.

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 supports three memory kinds corresponding to device-resident memory onboard the following popular GPU accelerator lines:

1. NVIDIA-branded GPUs with CUDA support
2. AMD-branded GPUs with ROCm/HIP support
3. Intel-branded GPUs with oneAPI/Level-Zero support

In the future it's anticipated that UPC++ will be extended to handle additional kinds of accelerator and non-host memories.

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 of presentation our examples use a single GPU device per process, however UPC++ can also handle nodes with multiple GPU devices or heterogeneous mixes of devices. See the *UPC++ Specification* for the details.

17.1   Data Movement between Host and GPU memory

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

To allocate storage, we first construct a device segment allocator using the `make_gpu_allocator()` factory function, which we then use to allocate shared objects in GPU memory. (This function was first made available in UPC++ release 2022.3.0 to simplify allocator management.) Of note, we pass the segment size to the allocator, in our case 4 MiB. The object is templated on the device type, and defaults to `gpu_default_device`. This type alias is set by compile definitions and usually indicates the type of GPU located on the machine being used. Valid device types include `upcxx::cuda_device`, `upcxx::hip_device` and `upcxx::ze_device`, corresponding to the three GPU hardware variants mentioned in the previous section. In this first example we explicitly request a device segment on a CUDA device. The `device_allocator` object returned here manages a CUDA shared memory segment of the specified size.

```
std::size_t segsize = 4*1024*1024;          // 4 MiB
// Allocate GPU memory segment
auto gpu_alloc = upcxx::make_gpu_allocator<upcxx::cuda_device>(segsize);
```

In addition to requesting a particular memory kind, function arguments are also available to request a specific CUDA device ID (for systems with multiple GPUs), or to optionally provide a pointer to previously allocated device memory. The full prototype is:

```
template <typename Device = gpu_default_device>
device_allocator<Device> make_gpu_allocator(size_t sz_in_bytes,
  Device::id_type device_id = Device::auto_device_id,
  void *device_memory = nullptr);
```

The `device_id` argument is an integer that defaults to a "smart" choice in the range `[0,Device::device_n())`, which is also the range of valid values to provide. In the case of no available GPUs at the calling process, the `device_allocator` returned from the function will be inactive. By default the function will allocate memory from the GPU driver library of the specified size to serve as the device segment.

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. Note that as with any memory allocator, a small amount of external padding may be inserted around allocated objects to ensure proper object alignment within the

segment. As such, device segments should be sized *slightly larger* than the sum of the anticipated data allocation sizes.

UPC++ also supports creation of device segments around pre-existing regions of device memory, for example a device memory area obtained from an on-node programming library, or one provisioned with non-default device-specific properties. The `device_memory` argument to `make_gpu_allocator()` supports this use case (not demonstrated in this example). If `device_memory` is specified, then `device_id` must correspond to the device where the memory resides, and must point to a segment of at least `sz_in_bytes` bytes. Any existing data in provided device memory will be preserved within the UPC++ segment, and the `device_allocator::to_global_ptr()` function can be used to obtain `global_ptr`'s to pre-existing device objects in such a segment.

The call to `make_gpu_allocator()` must be done collectively across all the processes. However different processes are permitted to pass different arguments. For example, processes may open devices with different IDs. If a given process wants to opt out of opening a device it can set the device ID to `invalid_device_id`; this is the default behavior of `auto_device_id` when called on a process that has no available devices. A different segment size can also be specified by each process. Once the `device_allocator` is constructed, allocation and deallocation of shared memory in the device segment is non-collective.

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);
```

Note the second template argument to the `global_ptr` which specifies the kind of target memory, i.e., `memory_kind::cuda_device`. This statically indicates a global pointer that references memory in a CUDA memory segment. This differs from pointers to host memory, where the second template argument to `global_ptr` defaults to `memory_kind::host`, indicating regular host memory. There is also a `memory_kind::any` kind which acts as a wildcard and generates a `global_ptr` type that can reference memory of any kind.

It's also worth noting that the `global_ptr` stack variables in this example could have been more concisely declared using the `auto` type specifier to leverage C++ type deduction; we've explicitly written out their types here to help clarify the presentation.

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

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

Data movement that potentially involves non-host memory 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 immediately wait on that future, but the full machinery of completions is available to enable overlap and schedule dependent operations (see Completions).

```
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.

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

This function will return the raw device pointer (a `T *`) 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 where nodes have multiple devices, it is important to obtain the ID of the device on which an allocation resides in order to properly launch a kernel. This can be retrieved using the `device_allocator::device_id()` method.

Here is a complete example:

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

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

using namespace std;
using namespace upcxx;

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

  std::size_t segsize = 4*1024*1024; // 4 MiB
  auto gpu_alloc = upcxx::make_gpu_allocator<cuda_device>(segsize); // alloc GPU segment
  UPCXX_ASSERT_ALWAYS(gpu_alloc.is_active(),
                      "Failed to open GPU:\n" << cuda_device::kind_info());

  // alloc some arrays 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();
  for (int i=0; i< 1024; i++) h1[i] = i; //initialize h1

  // 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 << "ERROR: " << nerrs << " errors detected" << endl;
  else if (!upcxx::rank_me()) cout << "SUCCESS" << endl;

  gpu_alloc.deallocate(gpu_array);
```

```
    delete_array(host_array2);
    delete_array(host_array1);

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

(see full example online)

17.2   RMA Communication using GPU memory

The power of the UPC++ memory kinds facility lies in its unified interface for moving data not only between host and device on the same node, but between source and target of any memory kind that reside on different nodes, by leveraging UPC++'s global pointer abstraction. We next show how to move data between a host and device that reside on different nodes. This time we will also use `gpu_default_device` for the device allocator instead of specifying `cuda_device`. As such, the code can be run without modifications using NVIDIA, AMD or Intel GPU hardware.

```
    upcxx::device_allocator<upcxx::gpu_default_device> gpu_alloc =
                                        upcxx::make_gpu_allocator(segsize);
```

Once we have set up a device allocator, we allocate host and device buffers as before, although this time the device memory `global_ptr` will have a memory kind of `gpu_default_device::kind`. Recall this is not the default template argument, so it needs to be specified (or deduced using `auto`). 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,gpu_default_device::kind>> dobj(gpu_array);
    int neighbor = (rank_me() + 1) % rank_n();
    global_ptr<double,gpu_default_device::kind> 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();
```

(see full example online)

Again, if we want to reverse the direction of the RMA data motion, we need only swap the first two arguments in the call to `copy()`. Instead of immediately waiting for it to complete, we could instead use completion objects such as the `operation_cx::as_rpc` mechanism mentioned in the Completions section. To modify the heat transfer simulation mentioned there for GPU compatibility, the memory buffers of both interior and halo cells could be allocated in device memory. This would mean that `copy()` operations between neighboring processes take the form of a device-to-device memory transfer. On compatible architectures, UPC++ can implement this using offload technologies, such as GPUDirect RDMA and ROCmRDMA. For a more thorough discussion and analysis of how to write a GPU version of the heat transfer example (which is decomposed in three dimensions and uses the Kokkos programming model for on-node parallelism), the reader is referred to this paper (doi:10.25344/S4630V).

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

```
    gpu_alloc.deallocate(gpu_array);
```

Note that in this second example, we have preceded this deallocation with a call to `upcxx::barrier` to prevent a process from deallocating the array from its device segment while a neighboring process may still be accessing it using `upcxx::copy()` (See Quiescence). Once all accesses to a device segment are quiesced, it can safely be destroyed using a collective call to the `device_allocator::destroy()` function:

```
   gpu_alloc.destroy();
```

In this section, we've shown how to use memory kinds in UPC++. This powerful feature relies on a unified notion of the global pointer abstraction to enable code using the pointer to interpret the kind of memory the pointer refers to, while hiding the details of the actual access method from the programmer. We've not demonstrated how to move data directly between GPUs, but the details follow the same patterns presented in the preceding examples.

Intermixing communication on device memory kinds with computational kernels on devices is straightforward, but GPU programming is beyond the scope of this guide. Please consult the UPC++ Extras package for several examples that demonstrate efficient use of the GPU with UPC++, as well as the gpu_vecadd example in the UPC++ distribution demonstrating vector addition.

18 Advanced Job Launch

The `upcxx-run` utility provides a convenience wrapper around a system-provided "native" job launcher, to provide a uniform interface for the most-common scenarios. However, for the "power user" the facilities of the native job launcher (such as core binding) are indispensable. Therefore, this section describes how to launch UPC++ applications with the native launcher.

The operation of `upcxx-run` includes the following three conceptual steps:

1. The UPC++ executable is parsed to extract necessary information, including the network for which it was compiled.
2. Necessary environment variables are set, for instance to implement the `-shared-heap` and `-backtrace` command line options.
3. A network-dependent next-level launcher is run.

To run a UPC++ application with the native launcher, you must first use `upcxx-run -v -show ...` to display the results of the three steps above (echoing the next-level launcher rather than executing it). Until you are familiar with the operation of `upcxx-run` for your scenario, take care to include all of options appropriate to your desired execution. If you cannot pass the intended executable, at least pass one compiled with the same installation and for the same network.

The following shows a simple example:

```
$ upcxx-run -v -show -n2 ./test-hello_upcxx-smp
UPCXX_RUN: ./test-hello_upcxx-smp is compiled with smp conduit
UPCXX_RUN: Environment:
UPCXX_RUN:    GASNET_MAX_SEGSIZE = 128MB/P
UPCXX_RUN:    GASNET_PSHM_NODES = 2
UPCXX_RUN:    UPCXX_SHARED_HEAP_SIZE = 128 MB
UPCXX_RUN: Command line:

    ./test-hello_upcxx-smp
```

The first line of the output above identifies the network as `smp`. The three lines indented below `Environment:` show the environment variables set by `upcxx-run` as well as any (potentially) relevant ones set in the environment prior to running `upcxx-run`. Finally, after `Command line:` we see the next-level launcher command (which is trivial in the case of `smp`).

The key to launching without `upcxx-run` is to run the native launcher with the same environment one would have if using it. In some cases, the command line in the `-show` output *is* the native launch command. In others it is a second-level wrapper provided by GASNet. The following sub-sections describe what to do with the command line, based on the network. However, since the presentation is incremental, you should read in order and *not* skip ahead to the network you are using.

18.1 Shared segment control

The UPC++ shared segment is created at job startup and used to service all shared heap allocations (see Global Memory). The shared segment size at each process is determined at job creation, as directed by the user's `upcxx-run -shared-heap` argument (see Running UPC++ Programs). This is currently implemented by setting the `UPCXX_SHARED_HEAP_SIZE` and `GASNET_MAX_SEGSIZE` environment variables, as one can see in the output above.

Specifying an appropriate shared heap size is important to the correct operation of many applications. However the details of how these variables interact is subtle (notably when passing a percentage argument) and subject to change. As such, we highly recommend that users wishing to bypass the `upcxx-run` wrapper first invoke `upcxx-run -v -show -shared-heap HEAPSZ` as suggested above with the actual desired HEAPSZ in order to obtain the appropriate values for these environment variables.

18.2   UPCXX_NETWORK=smp

In the case of the `smp` network, communication is performed through shared memory on a single host, and there is no actual network used. This leaves very little motivation to bypass `upcxx-run` in order to execute an application in this case. However, we present this case both for completeness and because it is the simplest case. Using the environment output shown prior to the start of this sub-section, one may launch a 2-process instance of `./test-hello_upcxx-smp` as follows:

```
$ env GASNET_MAX_SEGSIZE=128MB/P GASNET_PSHM_NODES=2 \
      UPCXX_SHARED_HEAP_SIZE='128 MB' ./test-hello_upcxx-smp
Hello from 0 of 2
Hello from 1 of 2
```

18.3   UPCXX_NETWORK=udp

The `udp` network case uses `amudprun` as its second-level launcher. The `-show` output may look like one of the following two examples depending on use of `-localhost` or `-ssh-servers`:

```
$ upcxx-run -show -v -n4 -localhost ./test-hello_upcxx-udp
UPCXX_RUN: ./test-hello_upcxx-udp is compiled with udp conduit
UPCXX_RUN: Looking for spawner "amudprun" in:
UPCXX_RUN:    /usr/local/pkg/upcxx/bin/../gasnet.opt/bin
UPCXX_RUN: Found spawner "/usr/local/pkg/upcxx/bin/../gasnet.opt/bin/amudprun"
UPCXX_RUN: Environment:
UPCXX_RUN:    GASNET_MAX_SEGSIZE = 128MB/P
UPCXX_RUN:    GASNET_SPAWNFN = L
UPCXX_RUN:    UPCXX_SHARED_HEAP_SIZE = 128 MB
UPCXX_RUN: Command line:

    /usr/local/pkg/upcxx/bin/../gasnet.opt/bin/amudprun -np 4 ./test-hello_upcxx-udp

$ upcxx-run -show -v -n4 -ssh-servers pcp-d-5,pcp-d-6,pcp-d-7,pcp-d-8 ./test-hello_upcxx-udp
UPCXX_RUN: ./test-hello_upcxx-udp is compiled with udp conduit
UPCXX_RUN: Looking for spawner "amudprun" in:
UPCXX_RUN:    /usr/local/pkg/upcxx/bin/../gasnet.opt/bin
UPCXX_RUN: Found spawner "/usr/local/pkg/upcxx/bin/../gasnet.opt/bin/amudprun"
UPCXX_RUN: Environment:
UPCXX_RUN:    GASNET_IBV_SPAWNFN = ssh
UPCXX_RUN:    GASNET_MAX_SEGSIZE = 128MB/P
UPCXX_RUN:    GASNET_SPAWNFN = S
UPCXX_RUN:    GASNET_SSH_SERVERS = pcp-d-5,pcp-d-6,pcp-d-7,pcp-d-8
UPCXX_RUN:    SSH_SERVERS = pcp-d-5,pcp-d-6,pcp-d-7,pcp-d-8
UPCXX_RUN:    UPCXX_SHARED_HEAP_SIZE = 128 MB
UPCXX_RUN: Command line:

    /usr/local/pkg/upcxx/bin/../gasnet.opt/bin/amudprun -np 4 ./test-hello_upcxx-udp
```

As with `smp`, running this `./test-hello_upcxx-udp` is simply a matter of running the indicated command line with the indicated environment. Taking the simpler `-localhost` case and folding-away `bin/../`:

```
$ env GASNET_MAX_SEGSIZE=128MB/P GASNET_SPAWNFN=L UPCXX_SHARED_HEAP_SIZE='128 MB' \
      /usr/local/pkg/upcxx/gasnet.opt/bin/amudprun -np 4 ./test-hello_upcxx-udp
Hello from 0 of 4
Hello from 1 of 4
Hello from 2 of 4
Hello from 3 of 4
```

18.4 UPCXX_NETWORK=aries, ibv or ofi

For the `aries`, `ibv` and `ofi` networks the output of `upcxx-run -v -show ...` will display a command line involving a next-level launcher `gasnetrun_[network]`. For instance (with line continuations added for readability):

```
$ upcxx-run -v -show -n4 -N2 ./test-hello_upcxx-aries
UPCXX_RUN: ./hello_upcxx is compiled with aries conduit
[... environment output elided ...]
UPCXX_RUN: Command line:

    <upcxx-install-path>/bin/../gasnet.opt/bin/gasnetrun_aries \
    -E UPCXX_INSTALL,UPCXX_NETWORK,UPCXX_SHARED_HEAP_SIZE \
    -N 2 -n 4 ./test-hello_upcxx-aries
```

Similar to `upcxx-run -show`, the `gasnetrun_[network]` utilities take `-t` to echo some verbose output which concludes with the command to be run (without running it). Continuing the example above (noting the `-t` inserted as the first argument to `gasnetrun_aries`):

```
$ env [environment-settings-from-upcxx-run] \
  [full-path-to]/gasnetrun_aries -t \
  -E UPCXX_INSTALL,UPCXX_NETWORK,UPCXX_SHARED_HEAP_SIZE \
  -N 2 -n 4 ./test-hello_upcxx-aries
[... some output elided ...]
gasnetrun: running: /usr/bin/srun -C knl -K0 -W30 -v -mblock \
              -n 4 -N 2 [full-path-to]/test-hello_upcxx-aries
```

In this case `/usr/bin/srun` is the native job launcher on a SLURM-based Cray XC system. Other native launchers one might see include `aprun`, `jsrun` or even `mpirun`.

Together with the environment from the initial `upcxx-run -v -show ...` the printed `/usr/bin/srun ...` command is sufficient to reproduce the behavior of the `upcxx-run` command without `-v -show`. With an understanding of the native launcher one can modify this command to suit one's needs, such as by addition of options to bind processes to CPU cores.

18.5 Single-step Approach

The preceding subsection describes a two-step process for determining the native launcher for certain networks. There is also a single-step approach: `upcxx-run -vv ...`. The output from both steps described above are produced in a single step, but with two key differences. The first is that this approach *will* run the provided executable (and thus may require you to run in an appropriate environment). The second is that in addition to the output required to uncover the native launcher, a *large* volume of additional verbose output is generated by the added execution.

18.6 Identifying and Avoiding ssh-spawner

The `ibv` network case may default to using a GASNet-provided "ssh-spawner". As the name implies, this utilizes `ssh` to perform job launch. This can be identified in the output of `gasnetrun_ibv -t ...` by the presence of `gasnetrun: set GASNET_SPAWN_ARGS=...`.

Since ssh-spawner bypasses the native job launcher, it is not suitable for obtaining more control over the job than is available via `upcxx-run`. You may try `GASNET_IBV_SPAWNER=mpi` or `GASNET_IBV_SPAWNER=pmi` in the environment to demand a different spawner. However, this may result in a failure from `gasnetrun_ibv` such as the following:
```
Spawner is set to PMI, but PMI support was not compiled in
```

18.7 Third-Level Launchers

In some cases, running `gasnetrun_[network] -t ...` may yield yet another level of launcher above the native one. When present, this is most often a site-specific wrapper intended to apply default behaviors (such as for core binding) or to collect usage statistics. Since there is no uniformity in such utilities, we cannot provide detailed instructions. However, there is often some way to enable a "show" or "verbose" behavior in order to find the native spawn command hidden beneath. Take care in such cases to seek out not only the final launch command, but also any environment variables which may be set to influence its behavior.

19   Additional Resources

Here are some helpful additional resources for learning and using UPC++:

Main UPC++ Site : upcxx.lbl.gov

- Software downloads
- Formal specification of UPC++ semantics
- Additional documentation
- Contact information

UPC++ Training Materials Site: upcxx.lbl.gov/training

- Video tutorials
- Hands-on exercises

UPC++ Frequently Asked Questions: upcxx.lbl.gov/faq

- Answers to questions about UPC++ from readers such as yourself

UPC++ Extras: upcxx.lbl.gov/extras

- Optional UPC++ extensions, implemented as libraries atop UPC++
- Extended UPC++ example codes

IPDPS'19 Paper: doi.org/10.25344/S4V88H

- Introductory peer-reviewed research paper
- Includes performance analysis of microbenchmarks and application proxies

UPC++ Issue Tracker: upcxx-bugs.lbl.gov

- Problem reports and feature requests

UPC++ User Support Forum: upcxx.lbl.gov/forum

- Questions and discussion regarding UPC++ programming

Thanks for your interest in UPC++ - we welcome your participation and feedback!