UPC++ Programmer's Guide (v2018.3.0)

Contents

1 Introduction **2**

2 Hello World in UPC++ **3**

3 Installing, Compiling and Running UPC++ Programs **4**

4 A Simple Example of Parallel Computation **5**

5 Global Memory **7**

6 Using Global Memory with One-sided Communication **8**

7 Asynchronous Computation **10**

8 Remote Procedure Calls **11**

9 Distributed Objects **12**

10 Conjoining Futures **13**

11 Atomics **15**

12 A Note on Performance **16**

13 Non-Contiguous One-Sided Communication **16**

14 Quiescence **18**

15 Completions **19**

16 Progress **20**

17 Personas **21**

18 View-Based Serialization **24**
18.1 The `view`'s Iterator Type 27
18.2 Buffer Lifetime Extension 28

19 Advanced Runtime Configuration **29**

Copyright

This manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher,

by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

Legal Disclaimer

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

Acknowledgments

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

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

---

1 Introduction

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

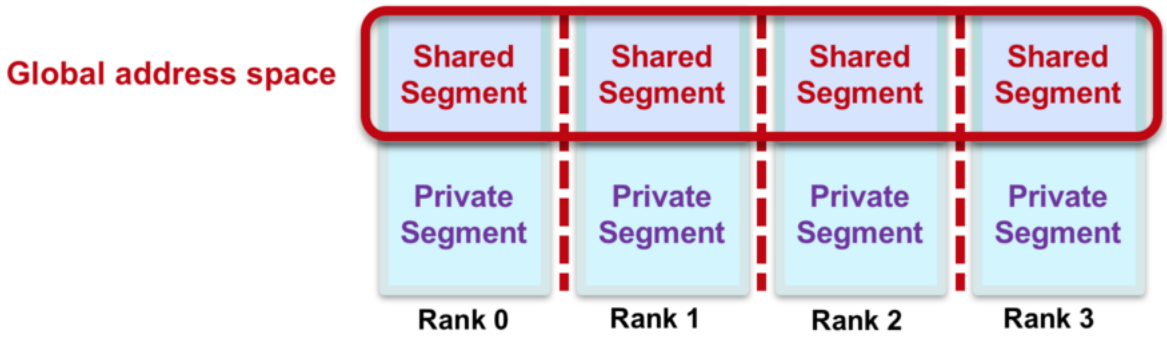

Figure 1. PGAS Memory Model.

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, low-level networking layer that provides network-independent, high-performance communication primitives tailored for implementing parallel global address space languages and libraries such as UPC, UPC++, Co-Array Fortran, Legion, Chapel, and many others. For more information about GASNet, visit http://gasnet.lbl.gov.

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

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

2 Hello World in UPC++

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

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

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

int main(int argc, char *argv[])
{
    // setup UPC++ runtime
    upcxx::init();
    // upcxx::rank_me() - get number for this rank
    cout << "Hello world from rank " << upcxx::rank_me() << endl;
    // close down UPC++ runtime
    upcxx::finalize();
    return 0;
}
```

All UPC++ programs need to be initialized with a call to `upcxx::init()` and finalized with a call to `upcxx::finalize()`. These calls set up and tear down the UPC++ runtime layer. `upcxx::init()` must be called before any UPC++ features are used, and no UPC++ features should be used after `upcxx::finalize()` is called (until the next call to `upcxx::init()`). Each UPC++ rank has a unique number (running from 0 to N-1, given N ranks), which can be accessed by a call to `upcxx::rank_me()`.

A UPC++ program is run with a fixed number of ranks, and it runs one copy of the program for each rank. In the Hello World example, this program will print out a message from each of the N ranks, for example, if N is 4, then the output could be as follows (note that there is no ordering enforced between the output from each rank):

```
Hello World from rank 2
Hello World from rank 0
Hello World from rank 3
Hello World from rank 1
```

3 Installing, Compiling and Running UPC++ Programs

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

Installing

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

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

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

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

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

To build for the compute nodes of a Cray XC, the `CROSS` environment variable needs to be set before the install command is invoked, i.e. `CROSS=cray-aries-slurm`. Additionally, because UPC++ does not currently support the Intel compilers (usually the default for these systems), either GCC or Clang must be loaded, e.g.:

```
module switch PrgEnv-intel PrgEnv-gnu
cd <upcxx-source-path>
CROSS=cray-aries-slurm ./install <upcxx-install-path>
```

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

The list of compatible versions of compilers for the various platforms can be found in the README.md that comes with the distribution, under the section "System Requirements". The `install` script checks that the compiler is supported and if not, it terminates with an error message indicating that `CXX` and `CC` need to be set to supported compilers, e.g. if a Mac has an old Homebrew install of gcc in `/usr/local/bin`, `CXX` and `CC` will need to be set to the latest Xcode versions in `/usr/bin`.

Compiling

To compile against UPC++, use the `<upcxx-install-path>/bin/upcxx-meta` helper script. This script takes a single parameter, one of `PPFLAGS`, `LDFLAGS` or `LIBFLAGS`, to properly set up the command line for compile and link stages:

- `PPFLAGS`: Preprocessor flags which will put the UPC++ headers in the compiler's search path and define macros required by those headers.
- `LDFLAGS`: Linker flags usually belonging at the front of the link command line (before the list of object files).
- `LIBFLAGS`: Linker flags belonging at the end of the link command line. These will make libupcxx and its dependencies available to the linker.

For example, to build the hello world code given previously, using `g++`, execute:

```
upcxx="<upcxx-install-path>/bin/upcxx-meta"
g++ --std=c++11 hello-world.cpp $($upcxx PPFLAGS) $($upcxx LDFLAGS) $($upcxx LIBFLAGS)
```

For an example, look at the `Makefile` in the `<upcxx-source-path>/example/prog-guide/` directory. That directory also has code for all of the examples given in this guide. To use the `Makefile`, first set the `UPCXX_INSTALL` shell variable to the install path.

UPC++ also supports multithreading within a rank, e.g. using OpenMP. In these cases, to ensure that the application is compiled against a thread-safe UPC++ backend, set `UPCXX_THREADMODE=par`. Note that this option is less efficient than the default, `UPCXX_THREADMODE=seq`, which enables the use of a UPC++ backend

that is synchronization free in most of its internals; thus, the parallel thread mode should only be used when multithreading within ranks.

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

```
export UPCXX_GASNET_CONDUIT=udp
```

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

Running

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

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

The launcher will run the executable and arguments `<exe> <args...>` in a parallel context with `<ranks>` number of UPC++ ranks.

Upon startup, each UPC++ rank creates a fixed-size shared memory heap that will never grow. By default, this heap is 128 MB per rank. This heap size can be adjusted 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 rank, call:

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

There are several options that can be passed to `upcxx-run`. Execute with `-h` to get a list of options.

4 A Simple Example of Parallel Computation

We illustrate parallel computation in UPC++ with a simple program that does a Monte Carlo calculation of `pi`. This contrived example was chosen because it provides a clear illustration of some of the properties of parallel computation, and has a known correct answer, so we can check our implementation. The value of `pi` can be calculated by repeatedly choosing a random point within the unit square, and counting the percentage of points that fall within the unit circle quadrant (see Figure 2). For a unit square with `r==1`, the area of the circle quadrant is `pi*r*r/4==pi/4`. A point x,y is inside the circle if `x*x+y*y<1`. So we can compute the ratio of the number of points inside the circle, `p_in`, to the total number of points, `p_tot`, in order to estimate `pi`, i.e. `pi=4*p_in/p_tot`.

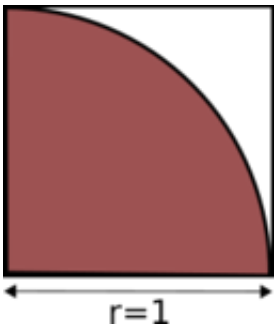

Figure 2. Computing `pi`.

In the program below, each rank calls a function `hit()` the same number of times (`my_trials`). The total amount of work done is proportional to `upcxx::rank_n()`, which gives the total number of ranks (thus, weak scaling). The `hit()` function returns 1 if a randomly chosen point falls within the unit circle quadrant and 0 otherwise. Thus each rank provides an independent estimate of `pi`.

The final step is a call to a function, `reduce_to_rank0`, which uses a UPC++ collective function, `upcxx::allreduce`, to sum all the separate results into a single value, so that rank 0 can finish the computation of `pi` and print out the result. Although all ranks will see the sum, only rank 0 needs it in the example; UPC++ currently does not support a reduce-to-one operation, which would store the result only in the root rank. Because the collective function is asynchronous, we have to synchronize on completion, using a to `wait()`. Later, in the Asynchronous Computation section, we will see how to overlap asynchronous operations, that is, when the wait is split-phased. But for now we do not separate the asynchronous call from the `wait()`. Note that the collective call also functions as a barrier, so we know that all ranks have completed their computations before we do the final sum.

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

using namespace std;

// choose a point at random
int64_t hit()
{
    double x = static_cast<double>(rand()) / RAND_MAX;
    double y = static_cast<double>(rand()) / RAND_MAX;
    if (x*x + y*y <= 1.0) return 1;
    else return 0;
}

// sum the hits to rank 0
// std::int64_t is used to prevent overflows
int64_t reduce_to_rank0(int64_t my_hits)
{
    // wait for a collective reduction that sums all local values
    return upcxx::allreduce(my_hits, plus<int64_t>()).wait();
}

int main(int argc, char **argv)
{
    upcxx::init();
    // each rank gets its own copy of local variables
    int64_t my_hits = 0;
    // the number of trials to run on each rank
    int my_trials = 100000;
    // each rank gets its own local copies of input arguments
    if (argc == 2) my_trials = atoi(argv[1]);
    // initialize the random number generator differently for each rank
    srand(upcxx::rank_me());
    // do the computation
    for (int i = 0; i < my_trials; i++) {
        my_hits += hit();
    }
    // sum the hits and print out the final result
    int64_t hits = reduce_to_rank0(my_hits);
    // only rank 0 prints the result
    if (upcxx::rank_me() == 0) {
        // the total number of trials over all ranks
```

```
        int64_t trials = upcxx::rank_n() * my_trials;
        cout << "pi estimate: " << 4.0 * hits / trials << ", "
             << "rank 0 alone: " << 4.0 * my_hits / my_trials << endl;
    }
    upcxx::finalize();
    return 0;
}
```

When the above code is executed with a small number of iterations-per-rank, for example:

```
upcxx-run -n 32 compute-pi 2
```

it produces an estimate for `pi` that is similar to the following (results will vary according to the random number generated employed):

```
pi estimate: 3.4375, rank 0 alone: 2
```

We can improve the estimate by increasing the number of ranks (64 and 128 ranks are shown):

```
pi estimate: 3.25, rank 0 alone: 2
pi estimate: 3.17188, rank 0 alone: 2
```

In the `allreduce` collective, `std::plus` was used to specify the arithmetic reduction operator. Unlike MPI, UPC++ does not provide a named set of built-in arithmetic operators for collective functions. Instead, UPC++ code is expected to used `std` functions, such as `std::plus` or `std::multiplies`, etc, or to define the operations as lambdas. An important caveat, however, is that only the `std` functions are eligible for hardware acceleration (should that exist for the given system); lambdas will never be accelerated, and so may not achieve the same performance.

5 Global Memory

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

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

The call to `upcxx::new_<int>` allocates a new integer in the calling rank's shared segment, and returns a global pointer (`upcxx::global_ptr`) to the allocated memory. This is illustrated in figure 3, which shows that each rank has its own private pointer (`gptr`) to an integer in its local shared segment. By contrast, a conventional C++ dynamic allocation (`int *mine = new int`) will be in private local memory. Note that we use the integer type in this paragraph as an example, but any type `T` can be allocated using the `upcxx::new_<T>()` function call.

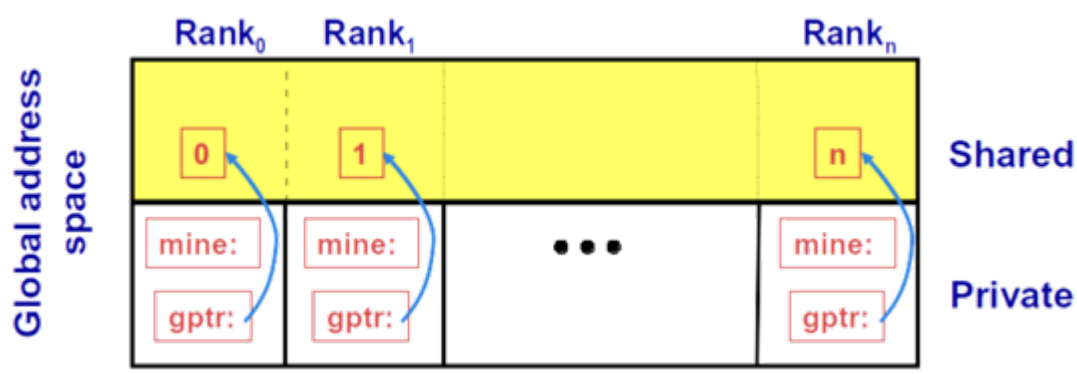

Figure 3. Global pointers.

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

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

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

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

In addition, UPC++ provides a function, `upcxx::new_array`, for allocating a 1-dimensional array in the global address space. Note that this array is not distributed as in UPC. Rather, each rank allocates its own array object which need not have the same size. The `upcxx::new_array` operation calls the *default* class constructor for the objects being allocated. The destruction operation is `upcxx::delete_array`. The function prototypes are:

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

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

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

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

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

6 Using Global Memory with One-sided Communication

We can now modify our code for computing `pi` to use global memory to get the total number of hits. The first step is for rank 0 to initialize a global pointer `all_hits_ptr` to a previously allocated 1D array, with one entry per rank, to hold all hits values from remote ranks. To allocate this array, we invoke the `upcxx::new_array` function. The pointer returned by the allocation is then broadcast to all ranks. Each rank adds an offset to this global pointer (the rank number) forming the new global pointer `my_hits_ptr`. Using this new global pointer, a rank can now put its local hits value to its own slot in the array, which is pointed to by `my_hits_ptr`. Since the array is in shared storage, every rank invokes the `upcxx::rput` function, a *remote put* to store the value at the target.

After it reaches the `upcxx::barrier`, rank 0 is permitted to sum all the values from remote ranks. Note the conversion of the global pointer `all_hits_ptr` to a local pointer, via the `upcxx::global_ptr<T>::local` function. This is possible because `all_hits_ptr` has affinity to rank 0, i.e. was allocated in the shared segment owned by rank 0. Finally, rank 0 deallocates the array pointed by `all_hits_ptr` using the `upcxx::delete_array` function.

```
int64_t reduce_to_rank0(int64_t my_hits)
{
    // Rank 0 creates an array the size of the number of ranks to store all
```

```
    // the global pointers
    upcxx::global_ptr<int64_t> all_hits_ptr = nullptr;
    if (upcxx::rank_me() == 0) {
        all_hits_ptr = upcxx::new_array<int64_t>(upcxx::rank_n());
    }
    // Rank 0 broadcasts the array global pointer to all ranks
    all_hits_ptr = upcxx::broadcast(all_hits_ptr, 0).wait();
    // All ranks offset the start pointer of the array by their rank to point
    // to their own chunk of the array
    upcxx::global_ptr<int64_t> my_hits_ptr = all_hits_ptr + upcxx::rank_me();
    // every rank now puts its own hits value into the correct part of the array
    upcxx::rput(my_hits, my_hits_ptr).wait();
    upcxx::barrier();
    // Now rank 0 gets all the values stored in the array
    int64_t hits = 0;
    if (upcxx::rank_me() == 0) {
        // get a local pointer to the shared object on rank 0
        int64_t *local_hits_ptrs = all_hits_ptr.local();
        for (int i = 0; i < upcxx::rank_n(); i++) {
            hits += local_hits_ptrs[i];
        }
        upcxx::delete_array(all_hits_ptr);
    }
    return hits;
}
```

The remote put function is part of the one-sided communication model supported by UPC++. Also supported is a remote get function, `upcxx::rget`. There are a number of variants of these two functions, the simplest being:

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

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

These operations initiate transfer of the `value` object to (put) or from (get) the remote rank; no coordination is needed with the remote rank (this is why it is *one-sided*). These operations return a completion object, which defaults to a future that becomes ready when the transfer is complete (futures are discussed in more detail in the Asynchronous Computation section. In our `reduce_to_rank0` example, we wait on the future before entering the `upcxx::barrier`, i.e.

```
upcxx::rput(my_hits, my_hits_ptrs).wait();
```

The type transferred must be serializable, in the sense of the C++ *trivially-copyable* concept. Support for serialization of more complex types will appear in a future release of Berkeley Lab UPC++.

Downcasting global pointers

In the example above, rank 0 gets the sum of the results put by remote ranks in the array pointed to by `all_hits_ptr`. This array is stored in shared memory with affinity to rank 0. Since the storage has affinity to the local rank, it is meaningful to downcast the global pointer to a local pointer, i.e. an ordinary C++ pointer. To this end, UPC++ provides the `local` method of `upcxx::global_ptr`:

```
int64_t *local_hits_ptrs = all_hits_ptr.local();
```

Using this downcast feature, we can treat all shared objects allocated by a rank as local objects (Only remote ranks need to use a global reference). Storage with local affinity can be accessed more efficiently via an ordinary C++ pointer. Note that for checking locality, UPC++ provides an `is_local` method for a global

pointer, which will return true if this global pointer is local and can be downcast. We need this call to avoid catastrophic errors, as it is is incorrect to attempt to downcast a global pointer that is *not* local to this rank. In the previous example, `all_hits_ptr` was allocated by rank 0 *only*:

```
if (upcxx::rank_me() == 0) {
    all_hits_ptr = upcxx::new_array<int64_t>(upcxx::rank_n());
}
```

And so if a rank other than rank 0 downcasts it, that would be an error, e.g. this is *incorrect*:

```
// WRONG - cannot downcast from a non-local global pointer!
int64_t *lp = all_hits_ptr.local();
```

Whereas this would be correct:

```
if (upcxx::rank_me() == 0) {
    int64_t *lp = all_hits_ptr.local();
}
```

Alternatively, a replicated directory of global pointers referencing shared objects allocated cyclically across all ranks can be set up as follows:

```
// declare a vector to store global pointers for all ranks
vector<upcxx::global_ptr<int64_t> > gps(upcxx::rank_n());
// allocate shared memory with affinity to this rank
gps[upcxx::rank_me()] = upcxx::new_<int64_t>();
// broadcast all global pointers to all ranks (gather-to-all)
for (int i = 0; i < upcxx::rank_n(); i++) {
    gps[i] = upcxx::broadcast(gps[i], i).wait();
}
// check for local pointer - should always be true
assert(gps[upcxx::rank_me()].is_local());
// now a rank can downcast its locally allocated global pointer
int *lp = gps[upcxx::rank_me()].local();
// and this is generally false
assert(gps[(upcxx::rank_me() + 1) % upcxx_rank_n()].is_local == false);
```

Having described the UPC++ constructs that deal with global storage and 1-sided communication, we next discuss about how to manage asynchronous communication, including remote procedure call. RPC is a powerful feature that enables the programmer to write highly scalable, performant code. Later, we'll see how they are an ingredient for writing communication tolerant code that overlaps communication with computation.

7 Asynchronous Computation

Most communication operations in UPC++ are asynchronous. So, in our original example, when we made the call to `upcxx::allreduce`, we had to explicitly wait 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 function prototype for `upcxx::allreduce` is (the UPC++ Specification includes a `upcxx::team` parameter, which we ignore since it is not yet supported):

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

The return type, `RType`, 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'll use this default completion here, and will return to the subject in detail in the Completions section. When the collective completes, the future becomes ready and can be used to access the results of the collective. The call to `wait()` in the `pi` estimation program can be replaced by the following equivalent polling loop, which exits when communication has completed:

```
upcxx::future<int64_t> my_hits_future = upcxx::allreduce(my_hits, plus<int64_t>());
while (!my_hits_future.ready()) upcxx::progress();
```

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. This common paradigm is embodied in the `wait()` method of `upcxx::future`.

Using futures, the rank waiting for a result can do computation while waiting, effectively overlapping computation and communication, e.g.:

```
upcxx::future<int64_t> my_hits_future = upcxx::allreduce(my_hits, plus<int64_t>());
// do unrelated work here
...
my_hits_future.wait();
```

An important feature of UPC++ is that there are no ordering guarantees with respect to asynchronous operations, i.e. 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, or errors will result.

8 Remote Procedure Calls

In our calculation of `pi`, instead of `upcxx::rput` or the `upcxx::allreduce` collective, we could use remote procedure calls (RPCs). An RPC enables the calling rank to invoke a function at a remote rank, using parameters sent to the remote rank via the RPC. The simplest prototype for the RPC call is:

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

This executes function `func` on rank `r` and returns the result in a future. The function passed in can be a lambda, or another function, but note that the function cannot be in a shared library. In the example below, we use a lambda in an RPC call to replace the `upcxx::allreduce` collective.

```
// need to declare a global variable to use with RPC
int64_t hits = 0;
int64_t reduce_to_rank0(int64_t my_hits)
{
    // wait for an rpc that updates rank 0's count
    upcxx::rpc(0, [](int64_t my_hits) { hits += my_hits; }, my_hits).wait();
    // wait until all ranks have updated the count
    upcxx::barrier();
    // hits is only set for rank 0 at this point, which is OK because only
    // rank 0 will print out the result
    return hits;
}
```

The lambda simply increments the global `hits` variable on rank 0. The work carried out in the RPC is done purely on rank 0, and the RPCs are serviced sequentially which ensures there is no possibility of a

race condition. Usually, this work is invoked by the UPC++ runtime inside calls to UPC++ functions. The mechanism is called *progress* and is described in more detail in the Progress section. Each rank waits for the RPC to complete (for the future to complete), and all ranks wait on a barrier (`upcxx::barrier()`), which means all ranks will have completed their updates before rank 0 computes and prints the final result.

The prototype for the barrier is (The UPC++ Specification includes a `upcxx::team` parameter, which we ignore since it is not yet supported):

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

In this specific case, the use of a global `upcxx::barrier()` is not necessary, as only rank 0 needs to be aware when all hits have been reduced. Indeed, rank 0 knows how many hits it is expecting, therefore a global variable `hits_counter` can be incremented within the RPC. Rank 0 can then poll on the value of `hits_counter` and call `upcxx::progress` until all hits have been received. Special care should be taken by the programmer when using lambda captures. In particular, *when passing C++ lambdas to the UPC++ RPC operations, reference captures should never be used.*

```
// need to declare a global variable to use with RPC
int64_t hits_counter = 0;
int64_t hits = 0;
int64_t reduce_to_rank0(int64_t my_hits)
{
    int64_t expected_hits = upcxx::rank_n();
    // wait for an rpc that updates rank 0's count
    upcxx::rpc(0,
               [](int64_t my_hits) {
                   hits += my_hits;
                   hits_counter++;
               },
               my_hits).wait();
    // wait until all ranks have updated the count
    if (upcxx::rank_me() == 0)
      while (hits_counter < expected_hits) upcxx::progress();

    // hits is only set for rank 0 at this point, which is OK because only
    // rank 0 will print out the result
    return hits;
}
```

9 Distributed Objects

UPC++ provides the concept of *distributed object*: a single logical object partitioned over a set of ranks (a team), where every rank 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:

```
upcxx::dist_object<int64_t> all_hits(upcxx:rank_me());
```

Each rank in a given team must call a constructor collectively for `upcxx::dist_object<T>`, with a value of type `T` representing the rank's instance value for the object (`upcxx::rank_me()` in the example above).

The need for universal distributed object naming stems primarily from RPC-based communication. If one rank needs to remotely invoke code on a peer's partition of a distributed object, there needs to be some mutually agreed-upon identifier for referring to that object.

In the `reduce_to_rank0` example below, the distributed object is an integer across all ranks, and the local instance of the object can be set as if it were a conventional pointer, as seen in the line `*all_hits =`

my_hits (a more compact version is to pass the value to the constructor, i.e. upcxx::dist_object<int64_t> all_hits(my_hits)). Although the constructor for a distributed object is collective, there is no guarantee that when the constructor returns on a given rank it will be complete on any other rank. To avoid this hazard, UPC++ provides an interlock to ensure that RPCs accessing a dist_object are delayed until the local representative has been constructed. However, this mechanism is necessary but not sufficient in our example. Because we are storing a value into the dist_object after construction, we need to insert a barrier to ensure that rank 0 won't access any remote instance of the distributed object until all the ranks have provided their contribution.

```
int64_t reduce_to_rank0(int64_t my_hits)
{
    // declare a distributed on every rank
    upcxx::dist_object<int64_t> all_hits(0);
    // set the local value of the distributed object on each rank
    *all_hits = my_hits;
    upcxx::barrier();
    int64_t hits = 0;
    if (upcxx::rank_me() == 0) {
        // rank 0 gets all the values
        for (int i = 0; i < upcxx::rank_n(); i++) {
            // fetch the distributed object from remote rank i
            hits += all_hits.fetch(i).wait();
        }
    }
    // ensure that no distributed objects are destructed before rank 0 is done
    upcxx::barrier();
    return hits;
}
```

To access the remote value of a distributed object, we use the dist_object::fetch member function, which given a rank argument, will send an RPC to get the T value from the sibling distributed object representative on the remote rank.

Note that due to the barrier mentioned above, the RPC is guaranteed to execute on the remote rank after the remote representative of the distributed object has been assigned the hits value. However, neither fetch nor any RPC referencing a dist_object require this synchronization - the RPC will automatically stall at the target as needed to await construction of each local representatives for all dist_object& references appearing as RPC arguments. This is precisely the interlock provided by UPC++ mentioned previously.

In the previous example, all the fetch 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. We illustrate how asynchrony can be used with a powerful UPC++ technique, *conjoining futures*, described in the next section.

10 Conjoining Futures

When many asynchronous operations are launched, it can be tricky to keep track of all the futures, and wait on all of them for completion. UPC++ provides an elegant solution, allowing futures to be *conjoined* (e.g. aggregated), so that the results of one future are dependent on others we need only wait for completion explicitly on that one future. The example below illustrates how the hits for the distributed objects can be accumulated asynchronously using this technique.

```
int64_t reduce_to_rank0(int64_t my_hits)
{
    // initialize this rank's part of the distributed object with the local value
```

```
    upcxx::dist_object<int64_t> all_hits(my_hits);
    int64_t hits = 0;
    // rank 0 gets all the values asynchronously
    if (upcxx::rank_me() == 0) {
        hits = my_hits;
        upcxx::future<> f = upcxx::make_future();
        for (int i = 1; i < upcxx::rank_n(); i++) {
          // construct the conjoined futures
          f = upcxx::when_all(f,
            all_hits.fetch(i).then([&](int64_t rhit) { hits += rhit; })
          );
        }
        // wait for the futures to complete
        f.wait();
    }
    upcxx::barrier();
    return hits;
}
```

The conjoining of futures in the code above can be viewed as an abstract graph, as shown in figure 4. The future conjoining begins with the construction of a trivially ready future on rank 0, using the `upcxx::make_future`. Rank 0 then loops through each remote rank and calls `fetch` asynchronously on each rank. The future returned by the `fetch` call has a *callback* or *completion handler* attached to it using the `.then` method of `upcxx::future`. The callback is executed on the initiating rank when the `fetch` completes, and is passed the result of the `fetch` as a parameter. In this example, the callback is a lambda, which adds the returned hits count for that rank, `rhits`, to the total hits, `hits`. Of course, the return value from the `.then` method call is itself another future. This future is passed to the `upcxx::when_all` function, in combination with the previous future, `f`. The `upcxx::when_all` constructs a new future representing readiness of all its arguments (in this case `f` and `.then`), and returns a future with a concatenated results tuple of the arguments. By setting `f` to the future returned by `when_all`, we can extend the conjoined futures. Once all the ranks are linked into the conjoined futures, rank 0 simply waits on the final future, i.e. the `f.wait()` call.

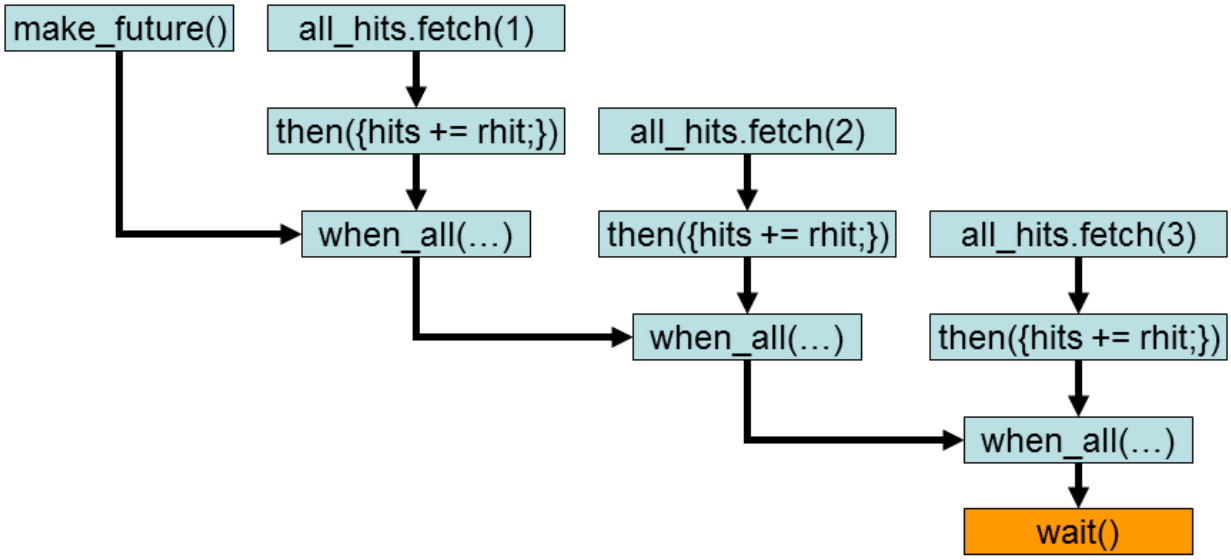

Figure 4. Graph of future conjoining.

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

- Only single future handle is needed to manage an entire logical operation and associated storage. The advantage is to simplify 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.

11 Atomics

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

Each atomic domain is an instance 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 independent of the data it applies to. User code is responsible for ensuring that data accessed via a given atomic domain is only accessed via that domain, never via a different domain or without use of a domain. 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 compute the `reduce_to_rank0` function in our ongoing example. In the code below, there is a single shared object allocated on rank 0, and all other ranks atomically increment it.

```
int64_t reduce_to_rank0(int64_t my_hits)
{
    // create the atomic domain
    upcxx::atomic_domain<int64_t> ad_i64({upcxx::atomic_op::load, upcxx::atomic_op::fetch_add});
    // a global pointer to the atomic counter in rank 0's shared segment
    upcxx::global_ptr<int64_t> hits_ptr =
        (!upcxx::rank_me() ? upcxx::new_<int64_t>(0) : nullptr);
    // rank 0 allocates and then broadcasts the global pointer to all other ranks
    hits_ptr = upcxx::broadcast(hits_ptr, 0).wait();
    // now each rank updates the global pointer value using atomics for correctness
    ad_i64.fetch_add(hits_ptr, my_hits, memory_order_relaxed).wait();
    // wait until all ranks have updated the counter
    upcxx::barrier();
    // once a memory location is accessed with atomics, it should only be
    // subsequently accessed using atomics to prevent unexpected results
    if (upcxx::rank_me() == 0) {
        return ad_i64.load(hits_ptr, memory_order_relaxed).wait();
    } else {
```

```
        return 0;
    }
}
```

Like all atomic operations (and indeed, nearly all UPC++ communication operations), atomic fetch and add operation is asynchronous. It returns a completion object, which defaults to a future, as shown by the prototype definition:

```
template <typename T, typename Completions=decltype(upcxx::operation_cx::as_future())>
RType upcxx::atomic_domain<T>::fetch_add(upcxx::global_ptr<T> p, T val,
                                         std::memory_order mo,
                                         Completions cxs=Completions{});
```

12 A Note on Performance

We have shown five different ways to get the result in the calculation of `pi`: reduction with collectives, RPCs, `rput` with global memory, distributed objects and atomics. These examples illustrate the use of the various options, but in practice, they would be used in different circumstances, taking performance into account. In our `reduce_to_rank0` examples, we expect the `upcxx::allreduce` to be the most efficient; all of the others essentially do a more expensive linear reduction, sometimes with contention.

13 Non-Contiguous One-Sided Communication

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

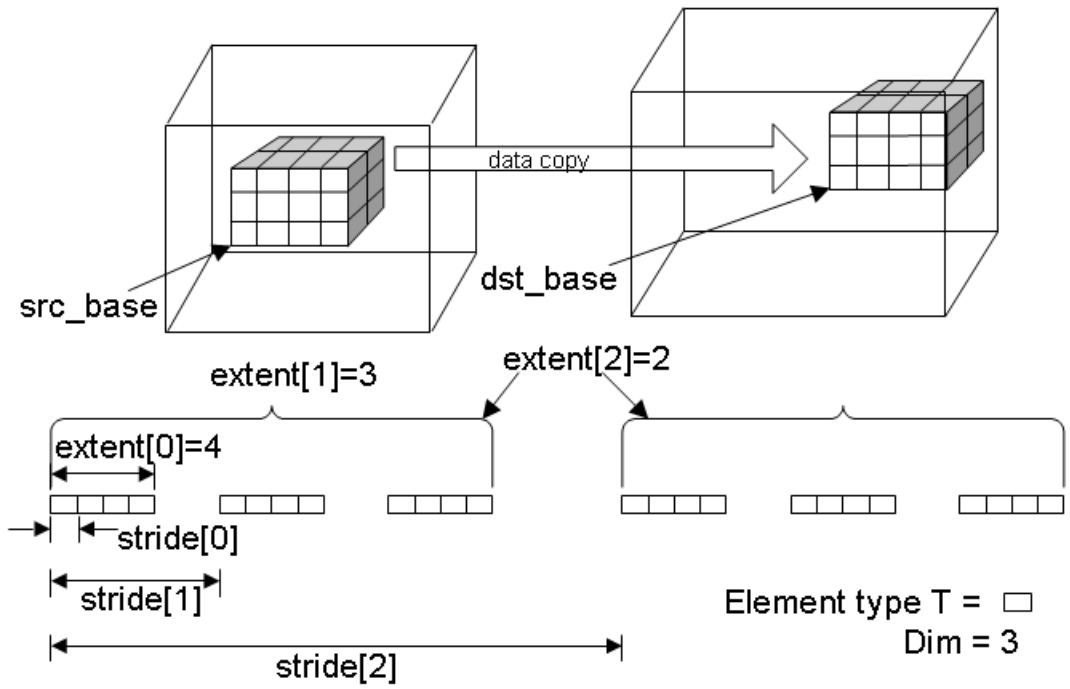

Figure 5. Example of strided one-sided communication.

```
constexpr int sdim[] = {32, 64, 32};
constexpr int ddim[] = {16, 32, 64};
future<> rput_strided_example(float* src_base, global_ptr<float> dst_base)
{
  return  rput_strided<3>(src_base,
          {{sizeof(float),sdim[0]*sizeof(float),sdim[0]*sdim[1]*sizeof(float)}},
                          dst_base,
          {{sizeof(float),ddim[0]*sizeof(float),ddim[0]*ddim[1]*sizeof(float)}},
                          {{4,3,2}});  }

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

The strided example code snippet corresponds to the translation data motion shown in Figure 5. By altering the arguments to the `_strided` functions a user can implement various transposes and reflection operations in an N-dimensional space within their UPC++ call. The striding is also expressed in units of bytes. This allows users to work with data structures that have 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<global_ptr<particle_t>, size_t> dest[]={{destP,38}};
auto f = rput_irregular(src, end(src), dest, end(dest));
f.wait();
```

The user here is taking subsections of their source array of their `particle_t` type pointed to by `srcP` and

copying them to the location pointed to by `destP`. `[rput,rget]` all assume the object types are trivially serializable. This example also shows how data can be shuffled in their UPC++ call. The user is responsible for ensuring their source and destination result in equivalent amounts of data.

14 Quiescence

Quiescence is a state in which no rank is performing computation that will result in communication injection, and no communication operations are currently in-flight in the network or queues on any rank. 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 `upcxx` computational phase.

To illustrate a simple approach to quiescence, we use the running example of computing `pi`. In this case, we use a version of RPC that does not return a future:

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

The "ff" stands for "fire-and-forget". From a performance standpoint, it has the advantage that is does not *send a response message* to satisfy the `future` back to the rank which has issued the RPC. However, because no acknowledgement is returned to the initiator, the caller does not get a future to indicate when the `upcxx::rpc_ff` invocation has completed execution at the target. The only way to determine completion is to use additional code. In using `upcxx::rpc_ff` in the `reduce_to_rank0` function, we need to add a counter, `n_done`, to track completion:

```
int64_t hits = 0;
// counts the number of ranks for which the RPC has completed
int n_done = 0;

int64_t reduce_to_rank0(int64_t my_hits)
{
    // cannot wait for the RPC - there is no return
    upcxx::rpc_ff(0, [](int64_t my_hits) { hits += my_hits; n_done++; }, my_hits);
    if (upcxx::rank_me() == 0) {
        // spin waiting for RPCs from all ranks to complete
        // When spinning, call the progress function to
        // ensure rank 0 processes waiting RPCs
        while (n_done != upcxx::rank_n()) upcxx::progress();
    }
    // wait until all RPCs have been processed (quiescence)
    upcxx::barrier();
    return hits;
}
```

We add a loop that spins waiting for the RPCs from all ranks to be completed. We know that rank 0 has to execute `upcxx::rank_n()` RPCs (issued by other ranks). The number of RPCs which have been executed is recorded by incrementing `n_done` in the body of the procedure issued by remote ranks (i.e. in the lambda function launched by the `upcxx::rpc_ff` call). As long as there are still RPCs to execute, rank 0 will call `upcxx::progress` to ensure that the upcxx runtime engine executes the RPCs being called on rank 0 (progress is described in more detail in the Progress section). Once `n_done` is equal to `upcxx::rank_n()`, rank 0 knows that it is now safe to exit, and can enter the `upcxx::barrier` on which all the other ranks are currently waiting.

There are multiple ways to achieve quiescence. When the number of messages to be received is known beforehand, it is possible to implement simple mechanisms such as the one used in our `reduce_to_rank0`

example. There are also more powerful (and thus more expensive) quiescence algorithms, such as the *counting algorithm*, that do not require the knowledge of the number of messages/RPCs beforehand. Advanced users requiring this capability should consult the appropriate literature.

15 Completions

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

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

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

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

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

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

Several of these completion concepts are demonstrated in the code below. It uses RPC completions, futures and promises. Every asynchronous `upcxx::rput` operation has an associated *RPC completion* which fulfills a *promise* on rank 0, which can be thought of as the *producer* side of the operation, whereas futures are the *consumer* side. In the previous examples of asynchronous calls in this guide, the UPC++ runtime would implicitly create a promise in each case. However, it is possible to explicitly create promises, which are a form of completion.

```
int64_t hits = 0;
upcxx::promise<> prom;
upcxx::global_ptr<int64_t> all_hits_ptr = nullptr;

int64_t reduce_to_rank0(int64_t my_hits)
{
    if (upcxx::rank_me() == 0)
        prom.require_anonymous(upcxx::rank_n());
    if (upcxx::rank_me() == 0)
        all_hits_ptr = upcxx::new_array<int64_t>(upcxx::rank_n());
    // Rank 0 broadcasts the array global pointer to all ranks
    all_hits_ptr = upcxx::broadcast(all_hits_ptr, 0).wait();
    // All ranks offset the start pointer of the array by their rank to point
    // to their own chunk of the array
    upcxx::global_ptr<int64_t> my_hits_ptr = all_hits_ptr + upcxx::rank_me();
    // operation completions
    auto cxs = (upcxx::operation_cx::as_future() |
```

```
                upcxx::remote_cx::as_rpc(
                    [](upcxx::intrank_t rank) {
                        hits += *(all_hits_ptr + rank).local();
                        prom.fulfill_anonymous(1);
                    }, upcxx::rank_me()));
    // all ranks try to write to their own part on rank 0 and then accumulate
    auto result = upcxx::rput(my_hits, my_hits_ptr, cxs);
    result.wait();
    upcxx::future<> fut = prom.finalize();
    fut.wait();
    if (upcxx::rank_me() == 0)
        upcxx::delete_array(all_hits_ptr);
    return hits;
}
```

Diving deeper into the details of the example above, rank 0 requires `upcxx::promise<> prom` to be fulfilled by all ranks. Then, it allocates an array to hold all the contributions from remote ranks, and broadcasts the obtained `upcxx::global_ptr`. Then, a *completion* `cxs` is created, requesting a RPC remote completion and a future operation completion for the subsequent `upcxx::rput` operation. When the `upcxx::rput` is launched, the sender is notified of its completion through the returned future `result`. This means that data can safely be discarded. On rank 0, when every incoming `upcxx::rput` operation are complete, a RPC gets executed. This RPC adds the incoming contribution to the global `hits` counter, and notifies `prom` that it has been fulfilled by one rank using `fulfill_anonymous`. When all ranks have executed their rput operation and the associated RPC completion, then `prom` will be satisfied on rank 0, and the associated future `fut` will become ready. That way, quiescence is achieved both on the sender side (using the future `fut`) and the receiver side (using RPC completion and `prom`).

16 Progress

Progress is a key notion of UPC++ which programmers must be aware of. The UPC++ framework does not spawn any hidden OS threads to advance its internal state or track outstanding asynchronous communication. The rationale is this keeps the performance characteristics of the UPC++ runtime as lightweight and configurable as possible. Without its own threads, UPC++ is reliant on each application rank 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(progress_level lev = 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, of note: `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 rank's local state to be in place.

Many UPC++ operations have a mechanism to signal completion to the application. However, for performance-oriented applications, UPC++ provides an additional asynchronous operation status indicator called `progress_required`. This status indicates that further advancements of the current rank 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 ranks will see their side of the completions without any further progress by the current rank. The programmer can query UPC++ when all operations initiated by this rank 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.

17 Personas

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

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's current/active status has changed threads when 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.

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 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 done with the `upcxx::persona_scope` constructor/destructor. It is built the following way:

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

The `upcxx::persona_scope` requires a thread locking mechanism (the `Lock` template argument can be any type of lock, such as C++ `std::mutex` for instance) and the `upcxx::persona` that needs to be pushed.

As an example, we show how to re-implement the computation of `pi` in a master-slave paradigm. Rank 0 is the master: it creates a persona (`scheduler_persona`), and issues RPCs to all the other ranks within the scope of `scheduler_persona`. Rank 0 then uses OpenMP threading to spawn an additional, secondary OpenMP thread, which has the sole purpose of making progress and accumulating the remote hits. Concurrently, the first OpenMP thread on rank 0 enters a computational phase during which it computes its own hits. Other ranks (slaves) call `upcxx::progress` until `done` is set to one by the RPC (See Quiescence). Note that in this example, we use OpenMP threads, but the example would work with any threading mechanism, such as pthreads or C++-11 threads.

```
#include <mutex>
#include <list>
#include <iostream>
#include <cstdlib>
#include <random>
#include <upcxx/upcxx.hpp>

using namespace std;

// choose a point at random
int64_t hit()
{
    double x = static_cast<double>(rand()) / RAND_MAX;
    double y = static_cast<double>(rand()) / RAND_MAX;
    if (x*x + y*y <= 1.0) return 1;
    else return 0;
}

int done = 0;

int main(int argc, char **argv)
{
    upcxx::init();
    srand(upcxx::rank_me());

    if (upcxx::rank_me() == 0) {
        // the number of trials to run on each rank
        int64_t trials_per_rank = 100000;
        if (argc == 2) trials_per_rank = atoi(argv[1]);
        int64_t hits = 0;
        int64_t my_hits = 0;

        upcxx::persona scheduler_persona;
        mutex scheduler_lock;
        list<upcxx::future<int64_t> > remote_rpcs;
        {
            // Scope block delimits domain of persona scope instance
            auto scope = upcxx::persona_scope(scheduler_lock, scheduler_persona);
            // All following upcxx actions will use scheduler_persona as current
            for (int rank = 1; rank < upcxx::rank_n(); rank++) {
                // launch computations on remote ranks and store the
                // returned future in the list of remote rpcs
                remote_rpcs.push_back(
                    upcxx::rpc(rank,
                               [](int64_t my_trials) {
                                   int64_t my_hits = 0;
                                   for (int64_t i = 0; i < my_trials; i++) {
                                       my_hits += hit();
                                   }
                                   done = 1;
                                   return my_hits;
                               },
                               trials_per_rank));
            }
```

```
            // scope destructs :
            // - scheduler_persona dropped from active set if it
            //   wasn't active before the scope's construction
            // - Previously current persona revived
            // - Lock released
        }
        #pragma omp parallel sections default(shared)
        {
            // This is the computational thread of rank 0
            #pragma omp section
            {
                // do the computation
                for (int64_t i = 0; i < trials_per_rank; i++) {
                    my_hits += hit();
                }
            } // end omp section
            // Launch another thread to make progress and track completion
            // of the operations
            #pragma omp section
            {
                auto scope = upcxx::persona_scope(scheduler_lock, scheduler_persona);
                while (!remote_rpcs.empty()) {
                    upcxx::progress();
                    auto it = find_if(remote_rpcs.begin(), remote_rpcs.end(),
                                      [](upcxx::future<int64_t> & f) {return f.ready();});
                    if (it != remote_rpcs.end()) {
                        auto &fut = *it;
                        // accumulate the result
                        hits += fut.result();
                        // remove the future from the list
                        remote_rpcs.erase(it);
                    }
                }
            } // end omp section
        } // end omp parallel sections
        hits += my_hits;
        // the total number of trials over all ranks
        int64_t trials = upcxx::rank_n() * trials_per_rank;
        cout << "pi estimated as " << 4.0 * hits / trials << endl;
    } else {
        // other ranks progress until quiescence is reached (i.e. done == 1)
        while (!done) upcxx::progress();
    }
    upcxx::finalize();
    return 0;
}
```

In the example above, the `hit()` function calls the C function `rand()` which is not thread-safe. However, this is not a bug for this particular example, because only one thread executes the *OpenMP section* in which the hit function is called. If multiple threads were to perform concurrent operations, the use of thread-safe routines would be required.

18 View-Based Serialization

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

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

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

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

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

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

For a more involved example, we will demonstrate one rank contributing histogram values to a histogram distributed over all the ranks. 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 rank is done by a hash function. We will demonstrate

transmission of the histogram update with and without view's, illustrating the performance advantages that views enable.

```
#include <upcxx/upcxx.hpp>
#include <string>
#include <unordered_map>

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

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

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

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

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

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

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

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

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

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

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

// Sending histogram updates by view.
```

```
upcxx::future<> send_histo1_byview(histogram1 const &histo) {
  std::unordered_map<upcxx::intrank_t, histogram1> clusters;

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

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

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

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

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

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

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

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

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

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

// This comparison functor orders keys such that they are sorted by
// owning rank at the expense of rehashing the keys in each invocation.
// A better strategy would be modify the map's key type to compute this
// information once and store it in the map.
```

```
struct histogram2_compare {
  bool operator()(std::string const &a, std::string const &b) const {
    using augmented = std::pair<upcxx::intrank_t, std::string const&>;
    return augmented(owner_of(a), a) < augmented(owner_of(b), b);
  }
};

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

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

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

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

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

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

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

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

    run_begin = run_end;
  }

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

18.1 The `view`'s Iterator Type

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

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

Notably, the view type has a second type parameter which is the type of its underlying iterator. If omitted,

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

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

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

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

18.2 Buffer Lifetime Extension

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

The following example demonstrates how a rank 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.

```
#include <upcxx/upcxx.hpp>
#include <array>
#include <vector>

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

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

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

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

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

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

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

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

19 Advanced Runtime Configuration

There are several environment variables that can be adjusted to fine-tune UPC++. When launching UPC++ jobs with `upcxx-run`, these variables are automatically set, depending on the parameters passed to the script. Two important parameters are:

- `UPCXX_SEGMENT_MB`: Set the shared memory heap per rank, which defaults to 128MB per rank. This value determines how much shared memory a rank can allocate before that allocation fails, and might need to be tuned on a per-run basis to control how much of the heap is reserved for shared memory versus how much is available for the private heap. The `-shared-heap` parameter to `upcxx-run` sets this value.
- `GASNET_MAX_SEGSIZE`: Set the underlying GASNet shared segment, which defaults to 2GB per rank on most systems. Note that this should always be strictly more than the `UPCXX_SEGMENT_MB` value. `upcxx-run` will ensure that this value is large enough for the `-shared-heap` setting, but it will never reduce the value from the default.