UPC++ Programmer's Guide (v1.0)

Contents

1 Introduction

UPC++ is a C++11 library that provides Asynchronous Partitioned Global Address Space (APGAS) programming. It is designed for writing parallel programs that run efficiently and scale well on distributed-memory parallel computers. The APGAS model is single program, multiple-data (SPMD), with each separate thread of execution (referred to as a *rank*, a term borrowed from MPI) having access to local memory as it would in C++. However, APGAS also provides access to a global address space, which is allocated in shared segments that are distributed over the ranks (see figure 1). UPC++ provides numerous methods for accessing and using global memory, as will be described later in this guide. In UPC++, all operations that access remote memory are explicit, which encourages programmers to be aware of the cost of communication and data movement. Moreover, all remote-memory access operations are by default asynchronous, to enable programmers to write code that scales well even on hundreds of thousands of cores.

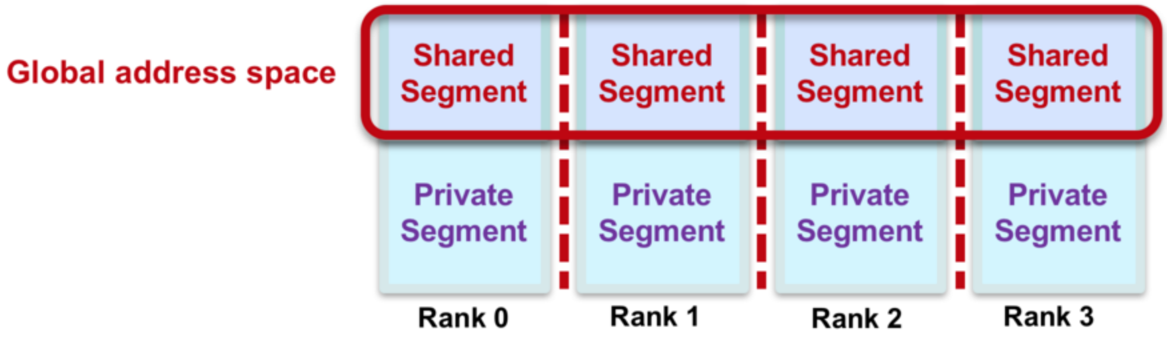

Figure 1: *APGAS Memory Model.*

This guide describes the Berkeley 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 this implementation of UPC++ uses GASNet, in this guide, only the Installing, Compiling and Running section is specific to the implementation. The Berkeley implementation of UPC++ adheres to the implementation-independent specification, which is available at the UPC++ homepage at https://bitbucket.org/berkeleylab/upcxx.

The code for UPC++ can be obtained from https://bitbucket.org/berkeleylab/upcxx. Please report any problems in the issue tracker.

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 code that implements 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:

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

Note that there is no ordering enforced between the output from each rank.

3 Installing, Compiling and Running UPC++ Programs

Presented here is 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. Users are recommended to use paths to non-existent or empty directories as the installation 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 aborts 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 takes a single parameter, one of `PPFLAGS`, `LDFLAGS` or `LIBFLAGS`:

- `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 running all of the examples given in the guide. To use the `Makefile`, first set the `UPCXX_INSTALL` shell variable to the install path.

Running

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

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

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

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`.

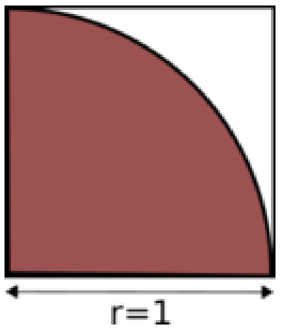

Figure 2: *Computing* `pi`*.*

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 (this is an example of 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 estimate `pi` and print out the result. The collective function is asynchronous, so we have to wait for the result (the call to `wait()`). The return value for asynchronous calls is described in the Asynchronous Computation section. 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
int 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
int reduce_to_rank0(int my_hits)
{
    // wait for a collective reduction that sums all local values
    return upcxx::allreduce(my_hits, plus<int>()).wait();
}

int main(int argc, char **argv)
{
    upcxx::init();
    // each rank gets its own copy of local variables
    int 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
    int 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
        int 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 32 compute-pi 2
```

It gives output similar to:

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

It can be seen that with low counts per rank, the estimate is poor for a single rank, whereas multiple threads improve the estimate.

In the `allreduce` collective, `std::plus` was used to define the arithmetic reduction operator. Unlike MPI, UPC++ does not provide explicit 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 Asynchronous Computation

Most communication operations in UPC++ are asynchronous. In the previous example, the collective call, `upcxx::allreduce`, is asynchronous, so we had to wait to get the result, using `wait()`. However, we can execute the wait at a later point, allowing us to overlap computation and communication. The function prototype for `upcxx::allreduce` is:

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

The return type is a UPC++ *future*, which holds a value (or tuple of values) and a state (ready or not ready). 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:

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

Note that the `upcxx::allreduce` has a parameter, `upcxx::team`. UPC++ supports *teams*, which are ordered sets of ranks. Collective operations apply to a team. The default team is `upxx::world()`, which includes every rank. Currently, this is the only team supported.

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.

6 Remote Procedure Calls

In our calculation of `pi`, instead of the `upcxx::allreduce` collective, we could use remote procedure calls (RPCs). An RPC enables a calling rank to send data plus a function to operate on that data to a remote rank. The prototype for the RPC call is:

```
template<typename Func, typename ...Args>
upcxx::future<R> upcxx::rpc(intrank_t r, F &&func, Args &&...args);
```

This executes function `func` on rank `r` and returns the result as a future of type `R`, which is (usually) the return type of `func`. The function passed in can be a lambda. This is how it is used in the example below, where we replace the `upcxx::allreduce` collective with an RPC using a lambda:

```
// need to declare a global variable to use with RPC
int hits = 0;
int reduce_to_rank0(int my_hits)
{
    // wait for an rpc that updates rank 0's count
    upcxx::rpc(0, [](int 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:

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

Like other collectives, the barrier applies to a team, which by default comprises all ranks.

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.

```
// need to declare a global variable to use with RPC
int hits_counter = 0;
int hits = 0;
int reduce_to_rank0(int my_hits)
```

```
{
    int expected_hits = upcxx::rank_n();
    // wait for an rpc that updates rank 0's count
    upcxx::rpc(0, [](int 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;
}
```

Special care should be taken by developers when using lambda captures.
Even in standard C++ functions within a single rank, lambda capture-by-reference (e.g. `[&foo](...)`) can be dangerous without careful attention to object lifetimes. UPC++ imposes several additional constraints:

- When a lambda expression is passed to UPC++ (for example, as an argument to `upcxx::persona::lpc`), a closure is created by C++ that contains captures of stack variables. That lambda closure is hidden from the calling program while it passes through the UPC++ library, making it difficult for developers to correctly manage the lifetime of any reference-captured variables.
- C++ reference captures are usually implemented by the C++ compiler using pointers. If a lambda closure containing reference captures is sent to another rank (for example, via `upcxx::rpc`) those pointers are no longer valid and very likely to cause memory corruption when the references are used at the target. (The same is also true for reference arguments or C++ pointers passed to an RPC function without proper serialization.)
- On distributed memory platforms, variables will often reside in various locations on different ranks due to non-symmetric stacks, making any use of capture-by-reference impossible.

Therefore, when passing C++ lambdas to UPC++ operations, reference captures are strictly prohibited.

7 Global Memory

A global pointer points to a shared object (which is an object allocated within a shared memory segment), 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 on 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 normal 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.

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 passing a pointer by value.

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 allocate a global pointer `all_hits_ptr` to an array so as to hold all hits values from remote ranks using the `upcxx::new_array` function. This pointer is then broadcast to all ranks, which offset this global pointer by their rank number and store the resulting global pointer in `my_hits_ptr`. Each rank then puts their local hits value to the space pointed by `my_hits_ptr` using the `upcxx::rput` function

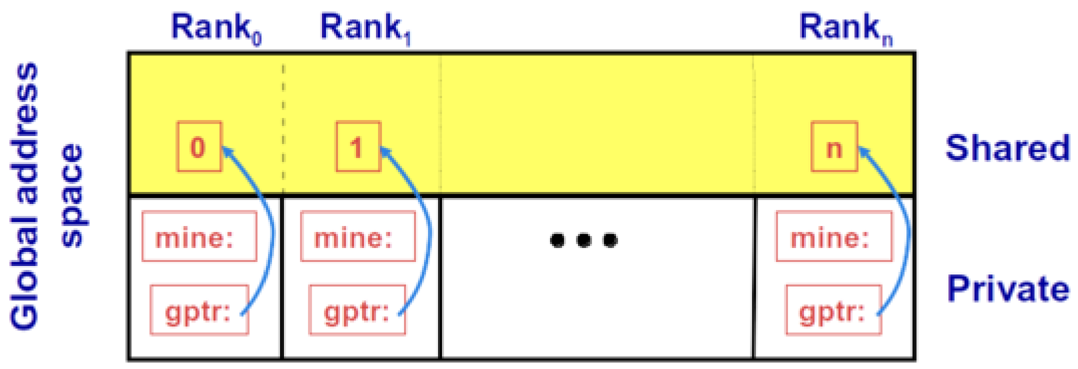

Figure 3: *Global pointers.*

(*remote put*). After hitting the `upcxx::barrier`, rank 0 can convert global pointer `all_hits_ptr` to a local pointer (using the `upcxx::global_ptr<T>::local` function), and sum all the values from remote ranks. Finally, rank 0 deallocates the array pointed by `all_hits_ptr` using the `upcxx::delete_array` function.

```
int reduce_to_rank0(int my_hits)
{
    // Rank 0 creates an array the size of the number of ranks to store all
    // the global pointers
    upcxx::global_ptr<int> all_hits_ptr = nullptr;
    if (upcxx::rank_me() == 0) {
        all_hits_ptr = upcxx::new_array<int>(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<int> 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
    int hits = 0;
    if (upcxx::rank_me() == 0) {
        // get a local pointer to the shared object on rank 0
        int *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>
future<> upcxx::rput(T value, upcxx::global_ptr<T> dest);
template<typename T>
```

```
future<T> upcxx::rget(upcxx::global_ptr<T> src);
```

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 future, which becomes ready when the transfer is complete. 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 be provided in subsequent versions.

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 rank 0's local memory, therefore the global pointer can be dereferenced to a local pointer using the `local` method of `upcxx::global_ptr`, as follows:

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

Using this feature, we can treat all shared objects allocated on a rank as local objects.

8 Allocating and Deallocating Memory in the Shared Segment

In our example above, we used the `upcxx::new_array` function to allocate an array of integers on rank 0. This function not only allocates shared objects, but calls the *default* class constructor for the objects being allocated. It is paired with `upcxx::delete_array` which calls the destructors. 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 single shared objects: `upcxx::new_` and `upcxx::delete_`. As with `upcxx::new_array_`, the `upcxx::new_` function calls the class constructor in addition to allocating memory. However, since it is a single object, arguments can be passed to the constructor, i.e. it does not have to be the default constructor. 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);
```

Finally, UPC++ 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);
```

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:

```
upcxx::dist_object<int> 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 (The rank's rank ID 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 agreeable identifier for referring to that object.

In the `reduce_to_rank0` example below, the distributed object is an integer across all ranks (the default `upcxx::world` team), and the local instance of the object can be set as if it is a regular 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<int> 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 in this case, we are assigning a value into the `dist_object` after construction, so we insert a barrier via `upcxx::barrier()` to ensure rank 0 won't access any remote instance of the distributed object until all the ranks have provided their contribution.

```
int reduce_to_rank0(int my_hits)
{
    // declare a distributed on every rank
    upcxx::dist_object<int> all_hits(0);
    // set the local value of the distributed object on each rank
    *all_hits = my_hits;
    upcxx::barrier();
    int 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 += fetch(all_hits, 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 define a `fetch` utility function, which takes the name of the distributed object and the rank we want to fetch. This function uses an RPC to get the instance of the distributed object of type `T` from the remote rank (we will use this convenience function several times in subsequent examples).

```
template <typename T>
upcxx::future<T> fetch(upcxx::dist_object<T> &dobj, upcxx::intrank_t rank) {
   return upcxx::rpc(rank, [](upcxx::dist_object<T> &rdobj) { return *rdobj; }, dobj);
}
```

Note that due 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 `fetch` does not require this synchronization - the RPC it uses will automatically stall at the target if needed to await construction of the local representative for a `dist_object` appearing as an RPC argument.

The getting of all the hits with distributed objects can also be done asynchronously, as shown below. In this example, we use chained futures to compute the results of the asynchronous `fetch` operations as they complete. All these operations are launched from rank 0. The chaining of futures starts with the construction of a trivially ready future, using the `upcxx::make_future` call, with the local value of `my_hits` on rank 0.

Then rank 0 loops through each remote rank, constructing the chain of futures, and then waits on the final combined future, `f`, for completion.

```
int reduce_to_rank0(int my_hits) {
    // initialize this rank's part of the distributed object with the local value
    upcxx::dist_object<int> all_hits(my_hits);
    int hits = 0;
    // rank 0 gets all the values asynchronously
    if (upcxx::rank_me() == 0) {
        upcxx::future<int> f = upcxx::make_future(my_hits);
        for (int i = 1; i < upcxx::rank_n(); i++) {
            // get the future value from remote rank i
            upcxx::future<int> remote_rank_val = fetch(all_hits, i);
            // create a future that combines f and the remote rank's result
            upcxx::future<int, int> combined_f = upcxx::when_all(f, remote_rank_val);
            // get the future for the combined result, summing the values
            f = combined_f.then([](int a, int b) { return a + b; });
        }
        // wait for the chain to complete
        hits = f.wait();
    }
    upcxx::barrier();
    return hits;
}
```

First, rank 0 fetches the remote value for the distributed object in a future, `remote_rank_val`. Instead of waiting for completion of the fetch as we did in our previous example at the beginning of this section, rank 0 combines `remote_rank_val` with the future `f` using the function `upcxx::when_all`, which constructs a future, `combined_f`, representing readiness of all the arguments, and returns a future with a concatenated results tuple of the arguments. We then call `.then` for the `combined_f` future, which allows us to attach a lambda to the results of the future, i.e. when the results are ready, the lambda executes. Note that the lambda takes as a parameter the result of the future, i.e. `future.result()`. At each loop, we chain a new future onto `f`.

Of course, the code within the for loop can be expressed much more succinctly, as shown below — we broke it down in the above example to make it easier to explain.

```
    // construct the chain of futures
    f = upcxx::when_all(f, fetch(all_hits, i)).then([](int a, int b) { return a + b; });
```

10 Atomics

UPC++ provides atomic operations on shared objects. This provides another mechanism for the `reduce_to_rank0` function in our ongoing example. Each atomic operation works on a global pointer to an approved atomic type, which are `std::int32_t`, `std::uint32_t`, `std::int64_t` and `std::uint64_t`. In the example below, there is a single shared object allocated on rank 0, and all other ranks atomically increment it.

```
int reduce_to_rank0(int my_hits)
{
    // a global pointer to the atomic counter in rank 0's shared segment
    upcxx::global_ptr<int32_t> hits_ptr =
        (!upcxx::rank_me() ? upcxx::new_<int32_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
    upcxx::atomic_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 upcxx::atomic_get(hits_ptr, memory_order_relaxed).wait();
    } else {
        return 0;
    }
}
```

The atomic fetch and add operation is asynchronous, like most UPC++ communication operations. It returns a future, as shown by the prototype definition:

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

The only other atomic operations currently supported are put and get:

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

template<typename T>
upcxx::future<T> upcxx::atomic_get(upcxx::global_ptr<T> p, memory_order mo);
```

11 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. The example illustrates 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.

12 Quiescence

Quiescence is a state in which ranks are not doing computations and no messages are currently being transferred on the network. 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(intrank_t receiver, F &&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 of this, there is no guarantee when the RPC will complete, so the only way to determine completion is using additional code. Using `upcxx::rpc_ff` in the `reduce_to_rank0` function, we need to add a counter, `n_done`, to track completion:

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

int reduce_to_rank0(int my_hits)
{
    // cannot wait for the RPC - there is no return
    upcxx::rpc_ff(0, [](int 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. We refer the advanced users requiring this capability to the appropriate literature.

13 Progress

Progress is a key notion of UPC++ which programmers should be aware of. The UPC++ framework does not use any private rank (thread) to advance its internal state and keep track of any outstanding asynchronous communication. Instead, UPC++ needs the application to give it access to the computing resource from time to time. To do so, UPC++ defines two levels of progress: *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 this level of progress. With user-level progress, UPC++ may advance its internal state as well as signal completion of user-initiated operations, such as RPCs.

It is very important for any programmer to understand that UPC++ needs to be given access periodically to the CPU. The `upcxx::progress` function used in our examples provides access to UPC++ explicitly.

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

`upcxx::progress` is the most important function to make user-level progress, and runs pending RPCs/callbacks on a particular rank. For the programmer, understanding which functions perform progress is crucial, since any invocation of user-level progress may execute RPCs or callbacks. When waiting on a future, user-level progress is also achieved.

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.

14 Personas

As mentioned earlier, UPC++ does not use background ranks for progressing asynchronous operations, but rather leaves all control to the user. The rationale is to make UPC++ as lightweight as possible. Instead of relying on its own internal ranks (threads), UPC++ introduces the concept of *personas*. An object of type `upcxx::persona` represents a state of the UPC++ library, and UPC++ manages a stack of active personas per OS thread. The top of that personas stack is referred to as the current persona. Note that a persona is current to *one and only one* OS thread at a time.

For any UPC++ operation issued by the current persona, the completion notification (e.g. future) will be sent to that same persona. This is still the case even if the active `upcxx::persona` object is not the current persona anymore by the time the asynchronous operation completes. The key takeaway here is that a `upcxx::persona` can be used by one rank to issue operations, then passed to another rank (together with the futures corresponding to these operations). That second rank 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 rank dedicated to progressing asynchronous operations.

UPC++ provides a `upcxx::persona_scope` class as a means of ensuring that only one OS thread is using a particular persona at a time. Pushing and popping personas from the persona stack (hence changing the current persona) is done with the `upcxx::persona_scope` type. 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 reimplement 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
int 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
        int trials_per_rank = 100000;
        if (argc == 2) trials_per_rank = atoi(argv[1]);
        int hits = 0;
        int my_hits = 0;

        upcxx::persona scheduler_persona;
        mutex scheduler_lock;
        list<upcxx::future<int> > 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,
                               [](int my_trials) {
                                   int my_hits = 0;
                                   for (int 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 (int 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<int> & 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
        int 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.