UPC++ Programmer's Guide

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.

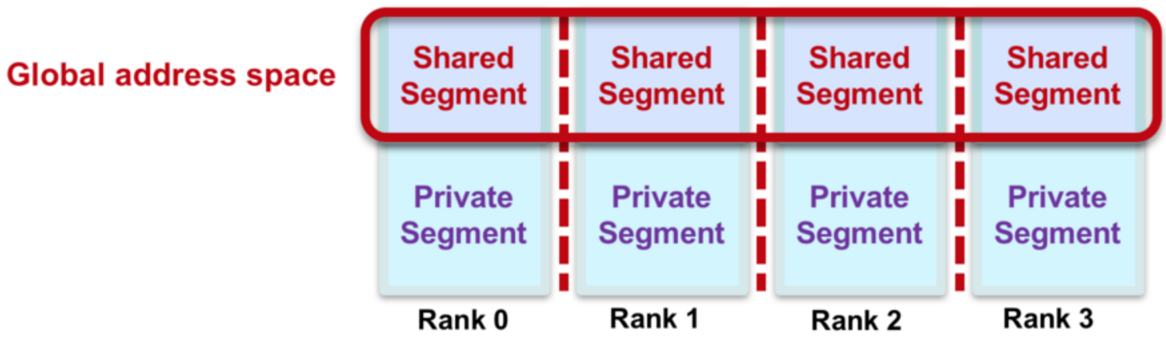

Figure 1: *APGAS Memory Model.*

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[])
{
    // this is the change!!!
    // 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

This programming guide assumes that you have obtained the source code file, `upcxx-v1.0-pre.tar.gz`.

To install, unpack the file and from the main directory run:

```
./install <installdirname>
```

This will install UPC++ to the `installdirname` directory.

If you are installing on Cray XC, before running install, further setup is needed:

```
export CROSS=cray-aries-slurm
export GASNET_CONDUIT=aries
module switch PrgEnv-intel PrgEnv-gnu
```

If there are any issues with the installation, it can be cleaned by running `rm -r .nobs`.

To compile, use the `${UPCXX_INSTALL}/bin/upcxx-meta` helper script, where `UPCXX_INSTALL` is the installation directory. For example, to build the hello world code given previously, execute:

```
g++ --std=c++11 hello-world.cpp $($UPCXX_INSTALL/bin/upcxx-meta PPFLAGS) \
    $($UPCXX_INSTALL/bin/upcxx-meta LDFLAGS) $($UPCXX_INSTALL/bin/upcxx-meta LIBFLAGS)
```

The compiled code can be run directly on an SMP node, but it will only run with one rank unless you set the number of ranks at run time. With the default SMP conduit (on a single node), this can be done with the `GASNET_PSHM_NODES` environment variable. So, for example, to run the hello world code on 8 ranks, use:

```
GASNET_PSHM_NODES=8 ./hello-world
```

For Cray XC with Slurm, the command for running hello world with 8 ranks would be:

```
srun -n 8  ./hello-world
```

For an example of a Makefile for building UPC++ applications, look at `example/prog-guide/Makefile`. This directory also has code for running all the examples given in this guide.

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`. 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`.

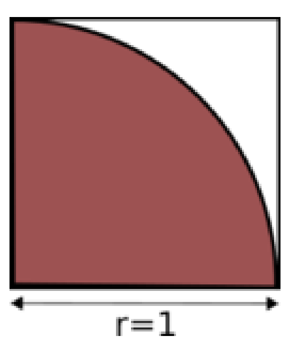

Figure 2: *Computing* ***pi****.*

In the program below, each rank makes an independent estimate of `pi`, i.e. there is no communication. 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, `accumulate`, 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 `upcxx::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;
}

// accumulate the hits
int accumulate(int my_hits)
{
    // wait for a collective reduction that sums all local values
    return upcxx::wait(upcxx::allreduce(forward<int>(my_hits), plus<int>() ));
}

int main(int argc, char **argv)
{
    upcxx::init();
    // each rank gets its own copy of local variables
    int my_hits = 0;
    // keep the number of trials per rank low to show the difference between
    // single and multiple ranks
    int my_trials = 2;
    // 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();
    }
    // accumulate and print out the final result
    int hits = accumulate(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 something like

```
GASNET_PSHM_NODES=32 ./compute-pi
```

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 overall it is better.

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 `upxx::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 sequence 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 `upcxx::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 `upcxx::wait` convenience function.

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
...
upcxx::wait(my_hits_future);
```

Note that the `upcxx::allreduce` has a parameter, `upcxx::team`. UPC++ supports *teams*, which are subgroups 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 receiver, 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 accumulate(int my_hits)
{
    // wait for an rpc that updates rank 0's count
    upcxx::wait(upcxx::rpc(0, [](int my_hits) { hits += my_hits; }, my_hits));
    // 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, which means that 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.

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. Please note that we use the integer type in this paragraph as an example, and any type `T` can be allocated using the `upcxx::new_<T>()` function call.

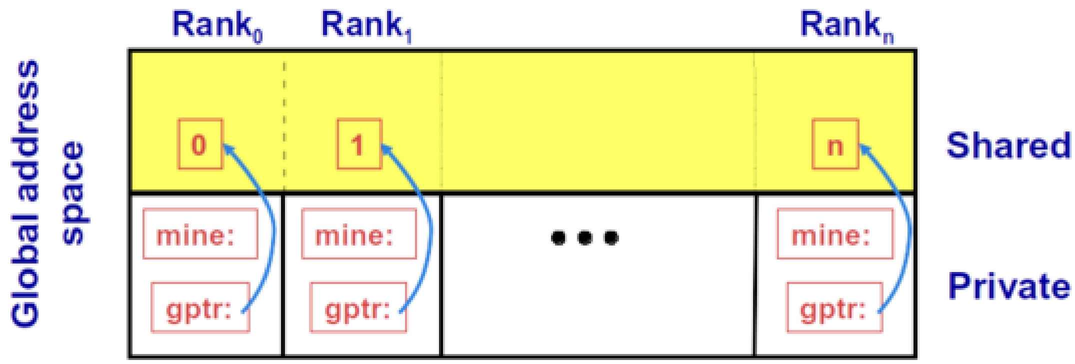

Figure 3: *Global pointers.*

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 for pointer arithmetic and passing a pointer by value.

We can now modify our code for computing `pi` to use global memory to accumulate the 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 (*remote get*). After hitting the `upcxx::barrier`, rank 0 can convert global pointer `all_hits_ptr` to a local pointer (using the upcxx::global_ptr::local function), and accumulate all values from remote ranks. Finally, rank 0 deallocates the array pointed by `all_hits_ptr` using the `upcxx::delete_array` function.

```
int accumulate(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::wait(upcxx::broadcast(all_hits_ptr, 0));
    // 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::wait(upcxx::rput(my_hits, my_hits_ptr));
    upcxx::barrier();
    // Now rank 0 accumulates 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 `accumulate` example, we wait on the future before entering the `upcxx::barrier`, i.e.

```
upcxx::wait(upcxx::rput(my_hits_ptrs[i]));
```

The type transferred must be serializable, in the sense of the C++ *trivially-copyable* concept. In future, support for serialization of more complex types will be provided.

In the example above, rank 0 accumulates 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 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_n());
```

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 current example.)

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 `accumulate` 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. Hence the call to `upcxx::barrier()` before rank 0 tries to access any remote instance of the object.

```
int accumulate(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 accumulates the values
        for (int i = 0; i < upcxx::rank_n(); i++) {
            // fetch the distributed object from remote rank i
            hits += upcxx::wait(fetch(all_hits, i));
        }
    }
    // 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 future examples). Note that the RPC will not be executed on the remote rank until the remote distributed object is finished construction, so the programmer does not have to check for this case.

```
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);
}
```

The accumulate 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 the distributed object 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 accumulate(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 accumulates 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 = upcxx::wait(f);
    }
    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 accumulate 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 accumulate(int my_hits)
{
    // a global point to the atomic counter in rank 0's shared segment
    // Only rank 0 allocates it and then it is broadcast to all other ranks
    upcxx::global_ptr<int32_t> hits_ptr =
        upcxx::wait(upcxx::broadcast(upcxx::new_<int32_t>(0), 0));
    // now each rank updates the global pointer value using atomics for correctness
    upcxx::wait(upcxx::atomic_fetch_add(hits_ptr, my_hits, memory_order_relaxed));
```

```
    // wait until all ranks have updated the counter
    upcxx::barrier();
    // once a global pointer is accessed with atomics, it should always be accessed
    // with atomics in future to prevent unexpected results
    if (upcxx::rank_me() == 0) {
        return upcxx::wait(upcxx::atomic_get(hits_ptr, memory_order_relaxed));
    } 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 accumulate 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 accumulation example, 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 `accumulate` 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 accumulate(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);
    // wait until all ranks have fired off the RPCs
    upcxx::barrier();
    if (upcxx::rank_me() == 0) {
        // spin waiting for all ranks to complete RPCs
        // When spinning, call the progress function to
        // ensure rank 0 processes waiting RPCs
        while (n_done != upcxx::rank_n()) upcxx::progress();
    }
    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 `accumulate` 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. Depending on the progress level, the application will be notified (if user-level progress has been asked for) of any update in UPC++'s internal state or not (internal progress).

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 thus run 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);
```

Finally, UPC++ provides the `upcxx::flush()` function to tell the UPC++ runtime that any delayed operation should be initiated. The `flush` operation also induces a `discharge`, and any application entering a long lapse of attentiveness (e.g. to perform expensive computations) is highly encouraged to call `upcxx::flush()` before.

```
void upcxx::flush();
```