# UPC++ as_eager Working Group Draft
# Revision 2020.6.2

Lawrence Berkeley National Laboratory Technical Report (LBNL-2001416)

Dan Bonachea

## Contents

**Acknowledgments**

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

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

**Revision History**

- June 2020: Revision 2020.6.1
  - Initial Working Group Draft, distributed to stakeholders.
- August 2021: Revision 2020.6.2
  - Minor updates to reflect implementation status
  - Lawrence Berkeley National Laboratory Technical Report LBNL-2001416

## 1 Introduction

This document is a working group draft proposing changes to the UPC++ library, to appear in a forthcoming version of the UPC++ specification and implementation.

This draft proposes an extension for a new future-based completion variant that can be more effectively streamlined for RMA and atomic access operations that happen to be satisfied at runtime using purely node-local resources. Many such operations are most efficiently performed synchronously using load/store instructions on shared-memory mappings, where the actual access may only require a few CPU instructions. In such cases we believe it's critical to minimize the overheads imposed by the UPC++ runtime and completion queues, in order to enable efficient operation on hierarchical node hardware using shared-memory bypass.

The new `upcxx::{source,operation}_cx::as_eager_future()` completion variant accomplishes this goal by relaxing the current restriction that future-returning access operations must return a non-ready future whose completion is deferred until a subsequent explicit invocation of user-level progress. This relaxation allows access operations that are completed synchronously to instead return a ready future, thereby avoiding most or all of the runtime costs associated with deferment of future completion and subsequent mandatory entry into the progress engine.

We additionally propose to make this new `as_eager_future()` completion variant the new default completion for communication operations that currently default to returning a future. This should encourage use of the streamlined variant, and may provide performance improvements to some codes without source changes. A mechanism is proposed to restore the legacy behavior on-demand for codes that might happen to rely on deferred completion for correctness.

Finally, we propose a new `as_eager_promise()` completion variant that extends analogous improvements to promise-based completion, and corresponding changes to the default behavior of `as_promise()`.

This document is a working group draft and is intended for readers who have some familiarity with C++ and the UPC++ Specification, Revision 2021.3.0. It is being distributed to UPC++ stakeholders to solicit feedback and convey design rationale. All of the extensions described herein are expected to continue evolving in response to stakeholder comments. All of the interfaces proposed in this document have now been implemented in the development version of UPC++ and made available in a 2021.3.6 snapshot of the library for early evaluation by users. For details on implementation status, please consult the ChangeLog distributed with the library implementation.

### 1.1 Providing Feedback

Readers are encouraged to provide feedback and comments on this working group draft via one of the following channels:

1. Comment on issue 107 in the UPC++ Specification tracker:
   https://bitbucket.org/berkeleylab/upcxx-spec/issues/107
   or alternatively open a new issue in that tracker.

2. Email to upcxx@googlegroups.com - this is a public support forum for users and maintainers of UPC++, and discussions are publicly visible.

3. Real-time video chat meetings with the Pagoda group's specification developers can be arranged upon request to pagoda@lbl.gov.

## 2 Overview:

`upcxx::local_team()` usually includes all UPC++ processes co-located on each physical node in a parallel job, and these processes communicate via interprocess shared-memory bypass rather than loopback network communication. `local_team()` provides the guarantee that global pointers referencing objects in the PGAS shared segment of team members may be down-cast to C++ pointers and accessed directly via regular language constructs. This leverages the fact that all node-local memory is physically load/store addressable by all local cores, regardless of process boundaries.

Optimizing for hierarchical systems in UPC++ often means leveraging `local_team()` to understand physical node boundaries and exploiting the more efficient communication available with node-local peers, for example to aggregate off-node communication operations or avoid unnecessary replication of data structures across the `local_team`. RMA operations on `global_ptr`'s referencing objects with affinity to `local_team` peers automatically use load/store bypass internally to synchronously perform the data transfer, however client synchronization for such operations still uses the regular UPC++ asynchronous completion machinery (e.g. `upcxx::future`). This machinery unfortunately incurs overheads which (while lightweight compared to off-node communication) can be significantly higher under current semantics than the cost of on-node accesses satisfied using load/store instructions via shared-memory bypass. These overheads are dominated by the cost of ensuring completion signaling for the operation is *deferred* until the next explicit user-level progress operation, and the cost of entering the user-level progress engine itself. It is this deferment which this proposal aims to eliminate where appropriate.

### 2.1 Manual Localization

For the use case of local RMA operations, these UPC++ communication overheads can be manually "bypassed" via static localization, where the `global_ptr<T>` is downcast to a raw `T*` and accessed using normal C++ pointer dereference operations, entirely bypassing UPC++ machinery for operations that are known to be most efficiently satisfiable via "synchronous" load/store instructions on the cache-coherent memory system (where they are also amenable to compiler and architectural reordering optimizations).

Manual localization works great for RMA cases where a `global_ptr<T> gptr` is *statically* known to point to memory with affinity to the `local_team()`. It's as simple as writing `*(gptr.local())` instead of calling `rput/rget`.
However it's less elegant for cases where `gptr` may or may not point to memory in the `local_team()` and we won't know which until runtime, but we still want to reap the benefits of reduced overheads. One obvious solution is dynamically checking for pointer locality, and then branching to an alternate version of the code performing the access(es).

#### 2.1.1 Example 1:

```
global_ptr<int> gptr = producer();  // gptr may or may not be local()
if (gptr.is_local()) {
   *(gptr.local()) = 42;
   do_something_dependent();
   do_something_overlappable();
} else {
   future<> f = rput(42, gptr);
   future<> f2 = f.then([] { do_something_dependent(); });
   do_something_overlappable();
   f2.wait()
}
```

### 2.1.2 Example 2:

```
// copies data from src->dst, regardless of affinity
// returns a future to signal completion of the copy
future<> copy_data(global_ptr<int> src, global_ptr<int> dst) {
  if (src.is_local()) {
     if (dst.is_local()) { // both local
        *(dst.local()) = *(src.local());
        return make_future(); // synchronously complete
     } else { // dst is remote
        return rput(*(src.local()), dst);
     }
  } else { // src is remote
     if (dst.is_local()) {
        return rget(src).then([dst](int val) { *(dst.local()) = val; });
     } else { // both remote
        return rget(src).then([dst](int val) { return rput(val, dst); });
     }
  }
}
```

### 2.1.3 Problems

This manual approach to dynamic localization is problematic for several reasons:

1. **It leads to unnecessary code duplication** Programmers end up laboriously writing two or more versions of the code, one for the case where a `global_ptr` input is local at runtime and another for when it's remote, decreasing productivity and maintainability. If there are N `global_ptr` inputs with independent locality properties (as in example 2), this can potentially expand to 2ˆN versions of the code, which is not scalable.
2. **The manual dynamic locality check is redundant with one that is always performed inside RMA calls** The programmer has manually incurred the cost of a branch (or N branches) to decide to use an RMA call vs downcast, but the RMA implementation already includes a (now redundant) locality branch to choose the correct RMA protocol (local load/store bypass vs remote network communication). In a code where the majority of the accesses are dynamically remote, the cost associated with these extra, redundant branches may add up to significant performance degradation.

To address this semantic gap, we propose the following `as_eager_future()` extension to the UPC++ completion object factories. It provides a mechanism to allow writing only one version of the code, while still dynamically avoiding many of the overheads of deferred completion and user-level progress for the case of shared accesses to on-node memory.

## 3 Proposed Specification for `as_eager_future`:

Here is a draft new API section for the completion chapter of the UPC++ specification:

```
[static] CType source_cx::as_eager_future();
[static] CType operation_cx::as_eager_future();
```

Constructs a completion object that represents notification of source or operation completion with a future.

This completion additionally permits the operation to return a future that is already signaled (i.e., `future<...>::ready()`) in cases where the completion occurred synchronously during initiation, rather than artificially delaying completion signaling of the future until a subsequent user-level progress.

UPC++ progress level: none

In most respects, this new completion variant behaves identically to the pre-existing `as_future()` variant. The only difference is that it grants communication initiation operations permission to return an already-ready future, to indicate synchronous satisfaction of the completion event. This is in contrast to the current specification which unconditionally prohibits return of ready futures from communication initiation operations.

### 3.0.1 Example 1: (rewritten)

```
  global_ptr<int> gptr = producer();  // gptr may or may not be local()
  future<> f = rput(42, gptr, operation_cx::as_eager_future());
  future<> f2 = f.then([] { do_something_dependent(); }); // callback might run now!
  do_something_overlappable();
  f2.wait();
```

In this example, if `gptr` happens to reference on-node memory (satisfying `is_local()`), the code above can (ideally) be implemented with no dynamic future allocation overheads and no progress overheads. The `rput` data transfer would execute as a synchronous store instruction on physically shared memory (as it already does), with the important difference being that the `rput` call is now permitted to return an already-ready future. We anticipate this empty "ready future" can be pre-allocated at runtime startup and re-used for all such cases, avoiding any allocation cost in the critical path. The `.then` operation invoked on a ready future will execute its callback synchronously (also returning a ready future) and the later `future::wait` operation would return immediately without entering the progress engine.

However when `gptr` happens to reference an off-node location, the *same* code starts the RMA, schedules the dependent callback and overlaps the overlappable code with the communication latency. The idea is the programmer can write *simpler* code but still reap most of the benefits of static localization - without maintaining multiple versions of the same algorithm or incurring redundant branches.

### 3.0.2 Example 2: (rewritten)

```
// copies data from src->dst, regardless of affinity
// permitted to return a ready future
future<> copy_data(global_ptr<int> src, global_ptr<int> dst) {
  return rget(src, operation_cx::as_eager_future())
             .then([dst](int val) {
                return rput(val, dst, operation_cx::as_eager_future());
              });
}
```

Here we've taken the 2ˆ2 versions of the code and removed the explicit locality branches, reducing it down to a single, non-branching expression. The hypothesis is that this (clearly more maintainable) code can still achieve similar performance when one or both locations are node-local at runtime. For the case that both locations are node-local, both RMA's are performed synchronously and this code never touches the user-level progress engine. The future result of the `rput` on a node-local address can be the canonical "ready future" (with no allocation overhead). The `rget` on a node-local address returns a trivially ready future, so with additional trickery it *might* eventually be possible to elide the allocation overheads for that future as well.

When one or both locations are actually remote, this expands to well-behaved asynchronous code at runtime and returns a non-ready future.

The proposed `operation_cx::as_eager_future()` completion would also be permitted for any other communication operations, notably including `atomic_domain` operations and non-contiguous RMA (i.e., `{rput,rget}_{strided,(ir)regular}`), which also often complete their work synchronously when invoked on node-local locations. We've omitted those examples for brevity.

## 4 Proposed Changes to default future completions

We believe the expected performance benefits from the proposed `as_eager_future` completion will motivate its use in most initiation calls using future completions. In order to encourage use of this new extension, we are additionally considering changing initiation calls which currently default to returning a deferred future to instead default to `as_eager_future` completion. This approach strongly encourages the use of the new alternative in codes with future-based asynchrony, automatically reducing overheads for future-based RMA on locations in `local_team` for many existing codes without source changes.

However, this follow-on proposal would be a potentially breaking change for some corner-case codes. Specifically this has the potential to break codes that issue future-based communication and then schedule callbacks on the future that cannot be safely run until the next user-level progress.

#### 4.0.1 Example 3:

```
global_ptr<int> gptr = ...;
int accum; // uninitialized
future<int> fut = rget(gptr);
fut.then([&accum](int result) { accum += result; });
accum = 0;
fut.wait();
```

This code currently works correctly because `fut` is guaranteed to be non-ready, so the callback cannot run until the next user-level progress. There are of course ways to rewrite all such codes to avoid requiring callback deferment (for example above, initializing `accum` earlier). However in some cases this might be non-trivial, especially if the assumption of callback deferment is implicit and undocumented.

### 4.1 Detailed Proposal

In order to deploy our desired change but still accommodate the possibility of such codes, we propose to:

1. Rename the current `{operation_cx,source_cx}::as_future()` completion object factories to `{operation_cx,source_cx}::as_defer_future()`, retaining the availability of the legacy future-deferment semantics.
2. Add new `{operation_cx,source_cx}::as_future()` completion object factories that by default are aliases for `{operation_cx,source_cx}::as_eager_future()`. `operation_cx::as_future()` remains the default completion variant for many communication operations (e.g. `rput`/`rget`), meaning these operations would now default to returning eager futures.
3. Add a user-defined macro override `UPCXX_DEFER_COMPLETION` that can be set to non-zero at the granularity of a translation unit before including `upcxx.hpp` (e.g. on the `upcxx` compile line) to restore the old default behavior, by having `as_future()` instead alias `as_defer_future()`.

This plan enables us to deploy the improved `as_eager_future` semantics for most users, while retaining a fallback option for existing codes that may depend on the legacy deferment behavior for correctness.

### 4.1.1 Examples 1 + 2: (rewritten)

Here's how the rewritten code now looks for example 1:

```
  global_ptr<int> gptr = producer();  // gptr may or may not be local()
  future<> f = rput(42, gptr);
  future<> f2 = f.then([] { do_something_dependent(); }); // callback might run now!
  do_something_overlappable();
  f2.wait();
```

and example 2:

```
// copies data from src->dst, regardless of affinity
// permitted to return a ready future
future<> copy_data(global_ptr<int> src, global_ptr<int> dst) {
  return rget(src).then([dst](int val) {
               return rput(val, dst);
             });
}
```

These codes no longer need to specify `operation_cx::as_eager_future()` explicitly, as it is now the default. They remain otherwise unchanged, but still reap the expected benefits of streamlined overheads for RMA on `local_team` locations. More importantly, this is true *even* if they were not originally written with potential locality in mind.

### 4.1.2 Example 3: (new release)

Compiling this code with `upcxx -DUPCXX_DEFER_COMPLETION` will restore deferred future completion as the legacy default, restoring correct behavior without code changes.

## 5 Proposal for `as_eager_promise`:

The previous sections have discussed optimizations to future-based completion, which is very common in many codes (especially those written for aggressive asynchrony). However UPC++ also supports several other completion mechanisms. Of these, the `as_promise` variant is notable in that it operates analogously to `as_future`, but manipulating a `upcxx::promise` which sits on the producer side of the synchronization handshake whereas a `upcxx::future` is on the consumer side.

`as_promise` completions are commonly used to efficiently aggregate several independent asynchronous communication completions into a single `future`, without the overhead of instantiating multiple explicit futures and conjoining them using `upcxx::when_all`. When using an `as_promise` completion, the client program passes an existing `promise` object to the communication initiation operation, which then increments the dependency counter on the promise to represent the asynchronous operation in-flight. Because the `promise` was provided by the client, the allocation for that object is managed explicitly by the client and (ideally) is amortized over many operations. As such, `as_promise` completions incur fewer allocation overheads at initiation than `as_defer_future` completions which require the runtime to dynamically allocate a new object to individually track each completion.

However, `as_promise` completions are still required to defer completion signaling (decrementing the dependency counter on the `promise`) until the next entry to user-level progress. This deferment semantic imposes some of the same costs associated with `as_defer_future`, namely requiring eventual explicit entry into the user-level progress engine to retrieve completion and process any dependent work which was also deferred.

For RMA operations on `local_team` locations that complete synchronously during initiation, this overhead is analogously unnecessary and potentially significant in relative cost.

## 5.1 Proposed Specification for `as_eager_promise`:

In order to streamline operations on `local_team` locations synchronized using promises, we propose the following draft addition to the API section for the completion chapter of the UPC++ specification:

```
template <typename ...T>
[static] CType source_cx::as_eager_promise(promise <T...> pro);
template <typename ...T>
[static] CType operation_cx::as_eager_promise(promise <T...> pro);
```

Precondition: `pro` must have a dependency count greater than zero.

Constructs a completion object that represents signaling the given promise upon source or operation completion.

In cases where the completion event occurred synchronously during initiation, this completion additionally permits the operation to signal completion immediately (by fulfilling the `promise` during the initiation call, resulting in zero net change to the dependency count) rather than artificially delaying fulfillment of the promise until a subsequent user-level progress.

UPC++ progress level: none

This new variant extends the same overhead reductions for `local_team` communication offered by `as_eager_future` to codes written with promise-based completion.

Note that due to the usual precondition requiring a non-zero dependency count for a promise passed to a communication initiation, synchronous fulfillment of the event in question cannot cause the promise to reach a "triggered" state during the initiation call. However a subsequent `promise::finalize()` call might return a ready future, where this previously could not occur without an intervening user-level progress to reap the deferred event signal from the communication call.

### 5.1.1 Proposed adjustment to `as_promise`:

Analogously to `as_future()`, we additionally propose to adjust the default behavior of existing `as_promise()` completions as follows:

1. Rename the current `{operation_cx,source_cx}::as_promise()` completion object factories to `{operation_cx,source_cx}::as_defer_promise()`, retaining the availability of the legacy deferred completion signaling semantics.
2. Add new `{operation_cx,source_cx}::as_promise()` completion object factories that by default are aliases for `{operation_cx,source_cx}::as_eager_promise()`. This means existing codes currently using `as_promise()` will now default to using eager promises.
3. Add a user-defined macro override `UPCXX_DEFER_COMPLETION` that can be set to non-zero at the granularity of a translation unit before including `upcxx.hpp` (e.g. on the `upcxx` compile line) to restore the old default behavior, by having `as_promise()` instead alias `as_defer_promise()`.

As with eager futures, this plan enables us to deploy the improved `as_eager_promise` semantics for existing promise-based codes without source changes, while preserving a fallback option for existing codes that may depend on the legacy deferment behavior for correctness.

## 6 Additional Resources

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

**Main UPC++ Site** : upcxx.lbl.gov

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

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

- Video tutorials
- Hands-on exercises

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

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

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

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

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

- Problem reports and feature requests

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

- Questions and discussion regarding UPC++ programming

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