Optimization of Asynchronous Communication Operations through Eager Notifications

Amir Kamil and Dan Bonachea

Applied Mathematics and Computational Research Division
Lawrence Berkeley National Laboratory

https://upcxx.lbl.gov/

Paper: doi:10.25344/S42C71

Video: doi:10.25344/S4JC7C
UPC++ overview

UPC++ uses a “Compiler-Free,” library approach
• UPC++ leverages C++ standards, needs only a standard C++ compiler

Relies on GASNet-EX for low-overhead communication
• Efficiently utilizes network hardware, including RDMA
• Provides Active Messages on which UPC++ RPCs are built
• Enables portability (laptops to supercomputers)

Designed for interoperability
• Same process model as MPI, enabling hybrid applications
• OpenMP and CUDA can be mixed with UPC++ as in MPI+X
What does UPC++ offer?

Asynchronous behavior

- **RMA: Remote Memory Access:**
  - Get/put/accumulate to a location in another address space
  - Low overhead, zero-copy, one-sided communication
- **RPC: Remote Procedure Call:**
  - Moves computation to the data

Design principles for performance

- All communication is syntactically explicit
- All communication is asynchronous: futures and promises
- Scalable data structures that avoid unnecessary replication
A Partitioned Global Address Space programming model

Global Address Space
- Processes may read and write *shared segments* of memory
- Global address space = union of all the shared segments

Partitioned
- *Global pointers* to objects in shared memory have an affinity to a particular process
- Explicitly managed by the programmer to optimize for locality
Asynchronous RMA in UPC++

By default, all communication operations are split-phased

- **Initiate** operation
- **Wait** for completion

```cpp
upcxx::global_ptr<int> gptr1 = ...;
upcxx::future<int> f1 =
    upcxx::rget(gptr1);

// unrelated work...
int t1 = f1.wait();
upcxx::future<> f2 =
    upcxx::rput(42, gptr1);
```

A UPC++ future holds values and a state: ready/not-ready

**wait** returns the result when the **rget** completes
Aggressive asynchrony via futures and callbacks

RMA returns a *future* object, which represents an operation that may or may not be complete.

Callbacks can be *chained* through calls to `then()`

Multiple futures can be *conjoined* with `when_all()` into a single future that encompasses all their results.

This code gets two remote values (an int and a double) and puts their product to another location:

```cpp
global_ptr<int>     source_i = ...;
global_ptr<double>  source_d = ...;
global_ptr<double>  target  = ...;
future<int>         fut1 = rget(source_i);
future<double>      fut2 = rget(source_d);
future<int, double> conj = when_all(fut1, fut2);
future<> res = conj.then([target](int a, double b) {
    return rput(a*b, target);
});
```
Completion: synchronizing communication

Communication can be synchronized using futures:

```cpp
future<int> fut = rget(remote_gptr);
int result = fut.wait();
```

This is just the default form of synchronization

- Most communication ops take a defaulted completion argument
- More explicitly: `rget(gptr, operation_cx::as_future());`
  - Requests future-based notification of operation completion

Other completion arguments may be passed to modify behavior

- Can trigger different actions upon completion, e.g.:
  - Signal a promise (the producer side of a future), deliver an RPC, etc.
  - Can even combine several completions for the same operation
Progress and deferred notifications

UPC++ does not spawn hidden threads to advance its internal state or track asynchronous communication

• Keeps the runtime lightweight and simplifies synchronization

Prior releases (2021.3.0 and earlier) required completion notifications to be deferred until the next call into the progress engine

• Provides consistent behavior for code such as:

```cpp
global_ptr<int> gptr = producer();
future<> f1 = rput(42, gptr);
future<> f2 = f1.then(... /* code block #1 */);
/* code block #2 */
f2.wait();
```

• Ensures that code block #2 executes before code block #1
Downsides of deferred notifications

Deferred notification can incur significant overheads for on-node accesses

- Future-based notification must allocate a promise cell on the heap, schedule it to be fulfilled later

Programmers often do manual localization to avoid this:

```cpp
global_ptr<double> gptr = ...;
if (gptr.is_local()) {
    *(gptr.local()) = 42; // direct load/store access
    // do overlappable computation
} else {
    future<> fut = rput(42, gptr);
    // do overlappable computation
    fut.wait();
}
```

- Leads to code bloat, duplicates locality check that is already in the runtime
Eager notifications

New eager notification added in 2021.3.6 snapshot, included in most recent 2021.9.0 release

• Immediately signals notification for synchronous completion

New factory methods for requesting deferred or eager notification:

   `operation_cx::as_defer_future()`
   `operation_cx::as_eager_future()`
   `operation_cx::as_defer_promise(promise<T...> &p)`
   `operation_cx::as_eager_promise(promise<T...> &p)`

New macro to control whether `as_future` and `as_promise` request eager or deferred notification

• If not defined, defaults to eager
Optimization of ready futures

Ready futures that do not encapsulate a value (i.e. `future<>`) are semantically equivalent with each other

- Implementation optimized to use common, pre-allocated internals

When conjoining multiple futures, if the resulting values and readiness only come from a single future, the result is semantically equivalent to that one input future

```cpp
future<int, double> fut1 = ... /* not ready */;
future<> fut2 = ... /* ready */, fut3 = ... /* ready */;
auto result = when_all(fut1, fut2, fut3);
```

Optimizations significantly improve performance of loops that conjoin many operations when most complete synchronously

```cpp
future<> f = make_future();
for (int i = 0; i < 10; ++i)
    f = when_all(f, rput(i, gptrs[i]));
```
Evaluation

Three versions of UPC++:
• 2021.3.0 release – most recent release prior to this work, used as control
• 2021.3.6 snapshot with deferred notifications
• 2021.3.6 snapshot with eager notifications

Benchmarks:
• Microbenchmarks: RMA and atomics
• GUPS: HPC Challenge RandomAccess benchmark
• Graph Matching: half-approximate maximum-weight matching

Experiments run on 3 systems on a single node, with 16 processes
• Only Intel Skylake results shown here; similar results on IBM Power9 and Marvell ThunderX2 (see paper)
Microbenchmarks

RMA or atomic transfers of 64-bit data between co-located processes

Each experiment timed 10M operations, initiating and then immediately waiting on each one

Average over 10 experiments

Observations:

• No performance regression between 2021.3.0 and 2021.3.6

• Eager is 46-92% faster than defer
Randomized fine-grained updates on distributed table

Several versions, using RMA or atomics, with future or promise notification

Observations:

• Eager is 3-15% faster than defer when using promises

• 147-304% faster when using futures due to skipping the progress engine as well as improvements to conjoining ready futures
Graph matching

Half-approximate maximum-weight matching from ExaGraph developers
Code optimizes updates to same process, but not to co-located processes

Experiments with four sparse graphs with varying degrees of locality from SuiteSparse Matrix Collection¹:

- Channel
- Delaunay
- Venturi
- Youtube

Additional graph randomly generated from the application itself, with ~13% of the edges between random vertices

Graph matching results

RMA-based UPC++ implementation by Sayan Ghosh (mel-upx)

Observations:

- Speedup limited by how much of the time is spent in communication, and what fraction is between different processes
- ~5% improvement for graphs with medium locality, 11% for graph with higher fraction of updates to co-located processes
Conclusions

The PGAS model enables the same code to operate on both on-node and off-node memory

- Provides productivity and maintainability

Asynchronous PGAS systems need to ensure that mechanisms for asynchrony only minimally impact performance of on-node operations

For UPC++, eager notifications provide significantly better performance than deferred notifications for on-node operations

- Up to 10x speedup for microbenchmarks, 3x for GUPS, 1.11x for graph matching on Intel Skylake
- Even higher speedups on other platforms (see paper)

Ongoing work in UPC++ to further optimize on-node operations
Acknowledgements

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation’s exascale computing imperative.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231, as well as This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Thank you!