UPC++: An Asynchronous RMA/RPC Library for Distributed C++ Applications

Damian Rouson
Berkeley Lab

The International Conference for High Performance Computing, Networking, Storage, and Analysis 2023 Tutorial

go.lbl.gov/sc23
UPC++: An Asynchronous RMA/RPC Library for Distributed C++ Applications

Damian Rouson

https://go.lbl.gov/SC23
pagoda@lbl.gov

Computer Languages and Systems Software (CLaSS) Group
Lawrence Berkeley National Laboratory
Berkeley, California, USA
Acknowledgements

This presentation includes the efforts of the following past and present members of the Pagoda group and collaborators:


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.
What does UPC++ offer?

Asynchronous behavior

- **RMA:**
  - Get/put to a remote 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
Reducing communication overhead

Let each process directly access another’s memory via a global pointer

Communication is **one-sided** – there is no “receive” operation

- No need to match sends to receives
- No unexpected messages
- No need to guarantee message ordering

All metadata provided by the initiator, rather than split between sender and receiver

- Supported in hardware through RDMA (Remote Direct Memory Access)

Looks like shared memory: shared data structures with asynchronous access
One-sided GASNet-EX vs one- and two-sided MPI

Two distinct network hardware types
The performance of one-sided GASNet-EX matches or exceeds that of MPI RMA and message-passing:
• 8-byte Put latency 35 - 46% better
• 8-byte Get latency 26 - 47% better
• Better flood bandwidth efficiency: often reaching same or better peak at ½ or ¼ the transfer size

8-Byte RMA Operation Latency (one-at-a-time)

Perlmutter Phase-I results collected July 2022, all others collected April 2023. GASNet-EX tests were run using then-current GASNet library and its tests. MPI tests were run using then-current center default MPI version and Intel MPI Benchmarks. All tests use two nodes and one process per node. For details see LCPC’18 doi.org/10.25344/S4QP4W and PAW-ATM’22 doi.org/10.25344/S40C7D
See also: gasnet.lbl.gov/performance
Global pointers

Global pointers are used to create logically shared but physically distributed data structures

Parameterized by the type of object it points to, as with a C++ (raw) pointer: e.g. `global_ptr<double>`, `global_ptr<Node>`
Global vs raw pointers and affinity

The affinity identifies the process that created the object.

Global pointer carries both an address and the affinity for the data.

Raw C++ pointers (e.g. Node*) can be used on a process to refer to objects in the global address space that have affinity to that process.
How does UPC++ deliver the PGAS model?

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 SPMD process model as MPI, enabling hybrid applications
- On-node compute models (e.g. OpenMP, CUDA, HIP, Kokkos) can be mixed with UPC++ as in MPI+X
Asynchronous communication (RMA)

By default, all communication operations are split-phased

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

A future holds a value and a state: ready/not-ready

```
global_ptr<int> gptr1 = ...;
future<int> f1 = rget(gptr1);
// unrelated work...
int t1 = f1.wait();
```

Wait returns the result when the rget completes
Remote procedure call (RPC)

Execute a function on another process, sending arguments and returning an optional result

1. Initiator injects the RPC to the target process
2. Target process executes \( fn(\text{arg1}, \text{arg2}) \) at some later time determined at the target
3. Result becomes available to the initiator via the future

Many RPCs can be active simultaneously, hiding latency

\[ \text{upcxx::rpc(target, fn, arg1, arg2)} \]

Execute \( fn(\text{arg1}, \text{arg2}) \) on process target
Compiling and running a UPC++ program

UPC++ provides tools for ease-of-use

Compiler wrapper:

$ upcxx -g hello-world.cpp -o hello-world.exe

- Invokes a normal backend C++ compiler with the appropriate arguments (-I/-L etc).
- We also provide other mechanisms for compiling
  - upcxx-meta
  - CMake package

Launch wrapper:

$ upcxx-run -N 1 -n 4 ./hello-world.exe

- Arguments similar to other familiar tools
- Also support launch using platform-specific tools, such as srun, jsrun and aprun.
Example: Hello world compile and run

Everything needed for the example codes is at: https://go.lbl.gov/SC23

Online materials include:
- Module info for NERSC Perlmutter, OLCF Frontier, and other machines
- Download links to install UPC++

Once you have set up your environment, copied the tutorial materials, and changed to the sc23/upcxx directory:

```bash
$ make run-hello-world
upcxx hello-world.cpp -Wall -o hello-world
upcxx-run -N 1 -n 4 ./hello-world
```

Hello world from process 2 out of 4 processes
Hello world from process 0 out of 4 processes
Hello world from process 3 out of 4 processes
Hello world from process 1 out of 4 processes

Copy this and change the number after `-n` to use a different number of processes, e.g.:

```bash
upcxx-run -N 1 -n 8 ./hello-world
```
Example: Hello world

```cpp
#include <iostream>
#include <upcxx/upcxx.hpp>
using namespace std;

int main() {
    upcxx::init();
    cout << "Hello world from process "
    << upcxx::rank_me()
    << " out of " << upcxx::rank_n()
    << " processes" << endl;
    upcxx::finalize();
}
```

Hello world from process 0 out of 4 processes
Hello world from process 2 out of 4 processes
Hello world from process 3 out of 4 processes
Hello world from process 1 out of 4 processes
Hello world with RPC (synchronous)

We can rewrite hello world by having each process launch an RPC to process 0

```cpp
int main() {
    upcxx::init();
    for (int i = 0; i < upcxx::rank_n(); ++i) {
        if (upcxx::rank_me() == i) {

            upcxx::rpc(0, [](int rank) {
                cout << "Hello from process " << rank << endl;
            }, upcxx::rank_me()).wait();
        }

    upcxx::barrier();
}

upcxx::finalize();
```

- **C++ lambda function**
- **Wait for RPC to complete before continuing**
- **Rank number is the argument to the lambda**
- **Barrier prevents any process from proceeding until all have reached it**
Futures

RPC returns a *future* object, which represents a computation that may or may not be complete.

Calling `wait()` on a future causes the current process to wait until the future is ready.

```cpp
upcxx::future<> fut = upcxx::rpc(0, [](int rank) {
    cout << "Hello from process " << rank << endl;
}, upcxx::rank_me());

fut.wait();
```
What is a UPC++ future?

A future is a handle to an asynchronous operation, which holds:

- The status/readiness of the operation
- The results (zero or more values) of the completed operation

The future is not the result itself, but a proxy for it.

The `wait()` method blocks until a future is ready and returns the result:

```cpp
upcxx::future<int> fut = /* ... */;
int result = fut.wait();
```

The `then()` method can be used instead to attach a callback to the future.
1D 3-point Jacobi in UPC++

Iterative algorithm that updates each grid cell as a function of its old value and those of its immediate neighbors

Out-of-place computation requires two grids

```c
for (long i = 1; i < N - 1; ++i)
    new_grid[i] = 0.25 *
        (old_grid[i - 1] + 2*old_grid[i] + old_grid[i + 1]);
```

Sample data distribution of each grid
(12 domain elements, 3 processes, N=12/3+2=6):
Jacobi boundary exchange (version 1)

RPCs can refer to static variables, so we use them to keep track of the grids

```c
double *old_grid, *new_grid;

double get_cell(long i) {
    return old_grid[i];
}
```

... 

```c
double val = rpc(right, get_cell, 1).wait();
```

We will generally elide the `upcxx::` qualifier from here on out.
Jacobi computation (version 1)

We can use RPC to communicate boundary cells

```cpp
future<double> left_ghost = rpc(left, get_cell, N-2);
future<double> right_ghost = rpc(right, get_cell, 1);

for (long i = 2; i < N - 2; ++i)
    new_grid[i] = 0.25 *
        (old_grid[i-1] + 2*old_grid[i] + old_grid[i+1]);

new_grid[1] = 0.25 *
    (left_ghost.wait() + 2*old_grid[1] + old_grid[2]);
new_grid[N-2] = 0.25 *

std::swap(old_grid, new_grid);
```

Initiate communication

Do interior computation

Wait for communication to complete and do boundary computation
One-sided put and get (RMA)

UPC++ provides APIs for one-sided puts and gets

Implemented using network RDMA if available – most efficient way to move large payloads

• Scalar put and get:

```cpp
global_ptr<int> remote = /* ... */;
future<int> fut1 = rget(remote);
int result = fut1.wait();
future<> fut2 = rput(42, remote);
fut2.wait();
```

• Vector put and get:

```cpp
int *local = /* ... */;
future<> fut3 = rget(remote, local, count);
fut3.wait();
future<> fut4 = rput(local, remote, count);
fut4.wait();
```
Jacobi with ghost cells

Each process maintains *ghost cells* for data from neighboring processes.

Assuming we have *global pointers* to our neighbor grids, we can do a one-sided put or get to communicate the ghost data:

```c
double *my_grid;
global_ptr<double> left_grid_gptr, right_grid_gptr;
my_grid[0] = rget(left_grid_gptr + N - 2).wait();
my_grid[N-1] = rget(right_grid_gptr + 1).wait();
```
Storage management

Memory must be allocated in the shared segment in order to be accessible through RMA

```cpp
global_ptr<double> old_grid_gptr, new_grid_gptr;
...
old_grid_gptr = new_array<double>(N);
new_grid_gptr = new_array<double>(N);
```

These are **not** collective calls – each process allocates its own memory, and there is no synchronization

- Explicit synchronization may be required before retrieving another process’s pointers with an RPC
- The pointers must be communicated to other processes before they can access the data
Downcasting global pointers

If a process has direct load/store access to the memory referenced by a global pointer, it can downcast the global pointer into a raw pointer with `local()`

```cpp
global_ptr<double> old_grid_gptr, new_grid_gptr;
double *old_grid, *new_grid;

void make_grids(size_t N) {
    old_grid_gptr = new_array<double>(N);
    new_grid_gptr = new_array<double>(N);
    old_grid = old_grid_gptr.local();
    new_grid = new_grid_gptr.local();
}
```

Downcasting can also be used to optimize for co-located processes that share physical memory
Jacobi RMA with gets

Each process obtains boundary data from its neighbors with \texttt{rget()}.

```cpp
future<> left_get = rget(left_old_grid + N - 2, old_grid, 1);
future<> right_get = rget(right_old_grid + 1, old_grid + N - 1, 1);
for (long i = 2; i < N - 2; ++i)
    /* ... */;
left_get.wait();
right_get.wait();
```

Begin asynchronous RMA gets
Overlapped computation on interior cells
Wait for communication, then consume values
Callbacks

The `then()` method attaches a callback to a future

- The callback will be invoked after the future is ready, with the future’s values as its arguments

```
future<> left_update =
    rget(left_old_grid + N - 2, old_grid, 1)
  .then([]() {
        new_grid[1] = 0.25 *
          (old_grid[0] + 2*old_grid[1] + old_grid[2]);
  });

future<> right_update =
    rget(right_old_grid + N - 2)
  .then([](double value) {
        new_grid[N-2] = 0.25 *
          (old_grid[N-3] + 2*old_grid[N-2] + value);
  });
```
Chaining calls using futures

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

```cpp
global_ptr<int> source = /* ... */;
global_ptr<double> target = /* ... */;
future<int> fut1 = rget(source);
future<double> fut2 = fut1.then([](int value) {
    return std::log(value);
});
future<> fut3 =
    fut2.then([target](double value) {
        return rput(value, target);
    });
fut3.wait();
```

This code retrieves an integer from a remote location, computes its log, and then sends it to a different remote location.
Conjoining futures

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

Can be used to specify multiple dependencies for a callback:

```cpp
global_ptr<int> source1 = /* ... */;
global_ptr<double> source2 = /* ... */;
global_ptr<double> target = /* ... */;
future<int> fut1 = rget(source1);
future<double> fut2 = rget(source2);
future<int, double> both = when_all(fut1, fut2);
future<> fut3 = both.then({[target](int a, double b) {
    return rput(a * b, target);
}});
fut3.wait();
```
Example: 2D heat diffusion

Everything needed for the example codes is at: https://go.lbl.gov/SC23

Online materials include download links to install UPC++

Once you have set up your environment, copied the tutorial materials, and changed to the sc23/upcxx directory:

```
$ make run-heat2d
upcxx heat2d.cpp -Wall -o heat2d
upcxx-run -N 1 -n 4 ./heat2d
```

[2] My Neighbors: (1, 3)  My Domain: (2048,3072)
[0] My Neighbors: (-1, 1)  My Domain: (0,1024)
[1] My Neighbors: (0, 2)  My Domain: (1024,2048)
[0] mean temperature=1.06256 | Solve time: 0.734826 seconds
2D heat diffusion data layout

Global (Abstract) View

Local (Concrete) View

Fixed boundary values

Process 2

Process 1

Process 0

Global (Abstract) View

Local (Concrete) View

\[ u_{i,j}^{n+1} = u_{i,j}^n + \alpha (u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n + u_{i,j+1}^n + u_{i,j-1}^n) \]

"Landing zone" for receiving data from downward neighbor

make run-heat2d
2D heat diffusion computation

Computation loop:

```cpp
for (int t = 0; t < num_timesteps; t++) {
    // initiate asynchronous puts to neighbors
    future<> fut = 
        when_all(rput(T_old, gptr_down, X),
                 rput(T_old+offset, gptr_up, X));

    // overlapped computation of interior
    compute_inner_T_new();

    // wait for my puts to complete
    fut.wait();
    // ensure everyone's puts have completed
    barrier();

    // compute boundaries using data received from neighbors
    compute_surface_T_new();

    // set up next timestep
    std::swap(T_new, T_old);
    barrier();
}
```

$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha (u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n + u_{i,j+1}^n + u_{i,j-1}^n)$$

make run-heat2d
Distributed objects

A *distributed object* is an object that is partitioned over a set of processes

```cpp
dist_object<T>(T value, team &team = world());
```

The processes share a universal name for the object, but each has its own local value

Similar in concept to a co-array, but with advantages

- Scalable metadata representation
- Does not require a symmetric heap
- No communication to set up or tear down
Example: Distributed hash table (DHT)

Distributed analog of \texttt{std::unordered\_map} (similar to Python \texttt{dict}, Java HashMap)

- Supports insertion and lookup
- We will assume the key and value types are \texttt{std::string}
- Represented as a collection of individual unordered maps across processes
- We use RPC to move hash-table operations to the owner

\begin{verbatim}
make run-dmap-insert-test
\end{verbatim}
DHT data representation

A distributed object represents the directory of unordered maps

class DistrMap {
    using dobj_map_t =
        dist_object<std::unordered_map<std::string, std::string>>;

    // Construct empty map
    dobj_map_t local_map{{}};

    int get_target_rank(const std::string &key) {
        return std::hash<std::string>{{key}} % rank_n();
    }
};
DHT insertion

Insertion initiates an RPC to the owner and returns a future that represents completion of the insert

```cpp
future<> insert(const string &key, const string &val) {
    return rpc(get_target_rank(key),
               [](dobj_map_t &lmap, const string &key, const string &val) {
                   (*lmap)[key] = val;
               }, local_map, key, val);
}
```

UPC++ uses the distributed object’s universal name to look it up on the remote process.
DHT find

Find also uses RPC and returns a future

```cpp
future<string> find(const string &key) {
    return rpc(get_target_rank(key),
        [](dobj_map_t &lmap, const string &key) {
            if (lmap->count(key) == 0)
                return string("NOT FOUND");
            else
                return (*lmap)[key];
        }, local_map, key);
}
```

UPC++ uses the distributed object's universal name to look it up on the remote process

Key passed as argument to the remote function

Send RPC to the process determined by key hash

Check whether key exists in local map

Retrieve corresponding value from the local map and return it

Process 0

Process p

Key passed as argument to the remote function

Send RPC to the process determined by key hash

Check whether key exists in local map

Retrieve corresponding value from the local map and return it

UPC++ uses the distributed object’s universal name to look it up on the remote process

Key passed as argument to the remote function
Optimized DHT scales well

Excellent weak scaling up to 32K cores [IPDPS19]
  • Randomly distributed keys

RPC and RMA lead to simplified and more efficient design
  • Key insertion and storage allocation handled at target
  • Without RPC, complex updates would require explicit synchronization and two-sided coordination

Cori @ NERSC (KNL)
Cray XC40
UPC++ advanced features

UPC++ has many advanced features that enable further optimizations

• Team-based barrier, reduction, and broadcast collectives
• Remote atomic operations that utilize hardware offload capabilities of modern networks
• Serialization of complex standard-library and user types in RPC’s
• Shared-memory bypass for co-located processes on many-core nodes
• Additional forms of communication completion notification such as promises and “signaling put”
• Non-contiguous RMA with automated packing and aggregation of strided or sparse data
• Memory kinds for data transfer between remote or local host (CPU) and device (e.g. GPU) memory
• …
UPC++ applications

UPC++ has been used successfully in several applications to improve programmer productivity and runtime performance, including:

- symPACK, a sparse symmetric matrix solver: PAW-ATM tomorrow
- SIMCoV, agent-based simulation of lungs with COVID
- MetaHipMer, a genome assembler
- Actor-UPCXX, used in the Pond tsunami simulator
- A UPC++ backend for NWChemEx/TAMM
- UPC++ DepSpawn, a library for data-flow computing
- Mel-UPX, half-approximate graph matching solver
SIMCoV: Spatial Model of Immune Response to Viral Lung Infection

Model the entire lung at the cellular level:

- 100 billion epithelial cells
- 100s of millions of T cells
- Complex branching fractal structure
- Time resolution in seconds for 20 to 30 days

**SIMCoV in UPC++**

- Distributed 3D spatial grid
- Particles move over time, but computation is localized
- Load balancing is tricky: active near infections

**UPC++ benefits:**

- Heavily uses RPCs
- Easy to develop first prototype
- Good distributed performance and avoids explicit locking
- Extensive support for asynchrony improves computation/communication overlap

[GitHub link](https://github.com/AdaptiveComputationLab/simcov)
ExaBiome: Exascale Solutions for Microbiome Analysis

What happens to microbes after a wildfire? (1.5TB)

What at the seasonal fluctuations in a wetland mangrove? (1.6 TB)

What are the microbial dynamics of soil carbon cycling? (3.3 TB)

How do microbes affect disease and growth of switchgrass for biofuels (4TB)

Combine genomics with isotope tracing methods for improved functional understanding (8TB)
Co-Assembly improves quality and is an HPC problem

Full wetlands data: 2.6 TB of data in 21 lanes (samples)
• Time-series samples from multiple sites of Twitchell Wetlands in the San Francisco Bay-Delta
• Previously assembled 1 lane at a time (multiassembly)
• MetaHipMer coassembled together – higher quality assembly, in 3.5 hours on 16K cores

This was the largest, high-quality de novo metagenome assembly completed at the time
More recently: new record 71TB metagenome assembly on 9000 nodes (288K processes and 72K GPUs)
of OLCF Frontier earlier this year.
UPC++ additional resources

Website: upcxx.lbl.gov includes the following content:

- Open-source/free library implementation
  - Portable from laptops to supercomputers
- Tutorial resources at upcxx.lbl.gov/training
- UPC++ Programmer’s Guide
- Videos and exercises from past tutorials
- Formal UPC++ specification
  - All the semantic details about all the features
- Links to various UPC++ publications
- Links to optional extensions and partner projects
- Contact information and support forum

“We found UPC++ to be a very powerful and flexible tool for the development of parallel applications in distributed memory environments that enabled us to reach the high level of performance required by our DepSpawn project, so that we could outperform the state-of-the-art approaches. It is also particularly important in our opinion that, while supporting a really wide range of mechanisms, it is very well documented and supported.”

-- Basilio Bernardo Fraguela Rodríguez, Universidade da Coruña, Spain

“If your code is already written in a one-sided fashion, moving from MPI RMA or SHMEM to UPC++ RMA is quite straightforward and intuitive; it took me about 30 minutes to convert MPI RMA functions in my application to UPC++ RMA, and I am getting similar performance to MPI RMA at scale.”

-- Sayan Ghosh, PNNL
Backup Slides
symPACK: UPC++ provides productivity + performance

Productivity
• RPC allowed very simple notify-get system
• Interoperates with MPI
• Non-blocking API

Reduced communication costs
• Low overhead reduces the cost of fine-grained communication
• Overlap communication via asynchrony/futures
• Increased efficiency in the extend-add operation
• Outperform state-of-the-art sparse symmetric solvers

https://upcxx.lbl.gov/sympack
Jacobi RMA with puts and conjoining

Each process sends boundary data to its neighbors with `rput()`, and the resulting futures are conjoined

```cpp
future<> puts = when_all(
    rput(old_grid[1], left_old_grid + N - 1),
    rput(old_grid[N-2], right_old_grid));

for (long i = 2; i < N - 2; ++i)
    /* ... */;

puts.wait();
barrier();
```

Ensure outgoing puts have completed
Ensure incoming puts have completed

```cpp
```
RPC and progress

Review: high-level overview of an RPC's execution
1. Initiator injects the RPC to the target process
2. Target process executes \( fn(arg1, \, arg2) \) at some later time determined at target
3. Result becomes available to the initiator via the future

*Progress* is what ensures that the RPC is eventually executed at the target
Progress

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

This design decision keeps the runtime lightweight and simplifies synchronization:

- RPCs are run in series on the main thread at the target process, avoiding the need for explicit synchronization.

The runtime relies on the application to invoke a progress function to process incoming RPCs and invoke callbacks.

Two levels of progress:

- Internal: advances UPC++ internal state but no notification.
- User: also notifies the application:
  - Readying futures, running callbacks, invoking inbound RPCs.
Invoking user-level progress

The `progress()` function invokes user-level progress

• So do blocking calls such as `wait()` and `barrier()`

A program invokes user-level progress when it expects local callbacks and remotely invoked RPCs to execute

• Enables the user to decide how much time to devote to progress, and how much to devote to computation

User-level progress executes some number of outstanding received RPC functions

• “Some number” could be zero, so may need to periodically invoke when expecting callbacks

• Callbacks may not wait on communication, but may chain new callbacks on completion of communication
Remote atomics

Remote atomic operations are supported with an *atomic domain*

Atomic domains enhance performance by utilizing hardware offload capabilities of modern networks

The domain dictates the data type and operation set

```
atomic_domain<int64_t> dom({atomic_op::load, atomic_op::min,
                              atomic_op::fetch_add});
```

- Supports all {32,64}-bit signed/unsigned integers, float, double

Operations are performed on global pointers and are asynchronous

```
global_ptr<int64_t> ptr = new<int64_t>(0);
future<int64_t> f = dom.fetch_add(ptr,2,memory_order_relaxed);
int64_t res = f.wait();
```
Serialization

RPC’s transparently \textit{serialize} shipped data

- Conversion between in-memory and byte-stream representations
- \texttt{serialize} $\rightarrow$ transfer $\rightarrow$ \texttt{deserialize} $\rightarrow$ invoke

Conversion makes byte copies for C-compatible types

- char, int, double, struct{double;double;}, ...

Serialization works with most STL container types

- vector<int>, string, vector/list<pair<int,float>>>, ...
- \textbf{Hidden cost}: containers deserialized at target (copied) before being passed to RPC function
Views

UPC++ views permit optimized handling of collections in RPCs, without making unnecessary copies

- **view**<T>: non-owning sequence of elements

When deserialized by an RPC, the **view** elements can be accessed directly from the internal network buffer, rather than constructing a container at the target

```cpp
vector<float> mine = /* ... */;
rpc_ff(dest_rank, [](view<float> theirs) {
    for (float scalar : theirs)
        /* consume each */
},
    make_view(mine)
);
```
Shared memory hierarchy and local_team

Memory systems on supercomputers are hierarchical
  • Some process pairs are “closer” than others
  • Ex: cabinet > switch > node > NUMA domain > socket > core

Traditional PGAS model is a “flat” two-level hierarchy
  • “same process” vs “everything else”

UPC++ adds an intermediate hierarchy level
  • local_team() – a team corresponding to a physical node
  • These processes share a physical memory domain
    • Shared segments are CPU load/store accessible across the same local_team
Downcasting and shared-memory bypass

Earlier we covered downcasting global pointers
- Converting `global_ptr<T>` from this process to raw C++ `T`
- Also works for `global_ptr<T>` from any process in `local_team()`

```cpp
int l_id = local_team().rank_me();
int l_cnt = local_team().rank_n();

global_ptr<int> gp_data;
if (l_id == 0) gp_data = new_array<int>(l_cnt);
gp_data = broadcast(gp_data, 0, local_team()).wait();
int *lp_data = gp_data.local();
lp_data[l_id] = l_id;
```

- Rank and count in my local node
- Allocate and share one array per node
- Downcast to get raw C++ ptr to shared array
- Direct store to shared array created by node leader
Optimizing for shared memory in many-core

`local_team()` allows optimizing co-located processes for physically shared memory in two major ways:

- Memory scalability
  - Need only one copy per `node` for replicated data
  - E.g. Cori KNL has 272 hardware threads/node
- Load/store bypass – avoid explicit communication overhead for RMA on local shared memory
  - Downcast `global_ptr` to raw C++ pointer
  - Avoid extra data copies and communication overheads
Completion: synchronizing communication

Earlier we synchronized communication 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 explicit: `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, inject an RPC, etc.
  - Can even combine several completions for the same operation

Can also detect other “intermediate” completion steps

- For example, source completion of an RMA put or RPC
Completion: promises

A promise represents the producer side of an asynchronous operation

- A future is the consumer side of the operation

By default, communication operations create an implicit promise and return an associated future

Instead, we can create our own promise and register it with multiple communication operations

```cpp
void do_gets(global_ptr<int> *gps, int *dst, int cnt) {
    promise<> p;
    for (int i = 0; i < cnt; ++i)
        rget(gps[i], dst+i, 1, operation_cx::as_promise(p));
    future<> fut = p.finalize();
    fut.wait();
}
```

Register an operation on a promise

Close registration and obtain an associated future
Completion: “signaling put”

One particularly interesting case of completion:

```cpp
callput(src_lptr, dest_gpgr, count,
    remote_cx::as_rpc([=]() {
        // callback runs at target rank after put data arrives
        compute(dest_gpgr, count);
    }));
```

- Performs an RMA put, informs the target upon arrival
  - RPC callback to inform the target and/or process the data
  - Implementation can transfer both the RMA and RPC with a single network-level operation in many cases
  - Couples data transfer w/sync like message-passing
  - BUT can deliver payload using RDMA without rendezvous (because initiator specified destination address)
Memory Kinds

Supercomputers are becoming increasingly heterogeneous in compute, memory, storage

UPC++ memory kinds enable sending data between different kinds of memory/storage media

API is meant to be flexible, but initially supports memory copies between remote or local CUDA GPU devices and remote or local host memory

```cpp
global_ptr<int, memory_kind::cuda_device> src = ...;
global_ptr<int, memory_kind::cuda_device> dst = ...;
copy(src, dst, N).wait();
```

Can point to memory on a local or remote GPU
GPU memory RMA on OLCF’s Frontier

Recent comparison of GASNet-EX and Cray MPICH performance on internode flood bandwidth benchmarks for six distinct combinations of (H)ost versus (G)PU memory and direction of transfer (Put or Get).

GASNet results were collecting using the testlarge benchmark that appears in the 2023.3.0 release.

MPI results are from osu_put_bw and osu_get_bw tests in a ROCM-enabled build of OSU Micro-Benchmarks 7.1-1.

All tests were run on OLCF Frontier in April 2023, between two nodes with one process per node, over its Slingshot-11 network.
Distributed objects in 2D heat diffusion

Distributed objects can be used to obtain global pointers to other processes’ landing zones

```cpp
global_ptr<double> down_in, up_in;
if (lo != 0) {
  down_in = new_array<double>(X);
  // ...
}
if (hi != Y) {
  up_in = new_array<double>(X);
  // ...
}
dist_object<global_ptr<double>> dist_up{down_in};
dist_object<global_ptr<double>> dist_down{up_in};
if (lo != 0) gptr_down = dist_down.fetch(down).wait();
if (hi != Y) gptr_up = dist_up.fetch(up).wait();
barrier();
```

Construct landing zones for each neighbor (if necessary)

Construct distributed objects containing pointers to each process’s landing zones

Fetch landing-zone pointer from the neighbor below

Ensure that all fetches have completed before the distributed objects are destroyed
Memory kinds: Accelerated RMA to/from GPU memory

Modern GPUs and NICs can support peer-to-peer data transfers

Example: Put with source on GPU

- In the absence of necessary hardware and OS support:
  1. Data must be copied from GPU memory to host memory
  2. RDMA from host memory’s copy

- With support:
  1. RDMA directly from GPU memory (no copies)
Memory kinds: Accelerated RMA to/from GPU memory

Measurements of flood bandwidth of upcxx::copy() on OLCF’s Summit

Difference between two consecutive releases shows benefit of GASNet-EX’s support for accelerated transfers via Nvidia’s “GDR”.

- No longer staging through host memory
- Large xfers: 2x better bandwidth
- Small xfers: up to 30x better bandwidth

Get operations to/from GPU memory now perform comparably to host memory

Comparisons to MPI RMA in GDR-enabled IBM MPI show UPC++ saturating more quickly to the peak

UPC++ results were collecting using the version of the cuda_benchmark test that appears in the 2020.11.0 release.

MPI results are from osu_get_bw test in a CUDA-enabled build of OSU Micro-Benchmarks 5.6.3.

All tests were run on OLCF Summit, between two nodes with one process per node, over its EDR InfiniBand network.
Non-contiguous RMA

We’ve seen contiguous RMA

• Single-element
• Dense 1-d array

Some apps need sparse RMA access

• Could do this with loops and fine-grained access
• More efficient to pack data and aggregate communication
• We can automate and streamline the pack/unpack

Three different APIs to balance metadata size vs. generality

• Irregular: iovec-style iterators over pointer+length
• Regular: iterators over pointers with a fixed length
• Strided: N-d dense array copies + transposes