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

Paul H. Hargrove

https://upcxx.lbl.gov
pagoda@lbl.gov

Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California, USA
Background and motivation
The Pagoda project

Support for lightweight communication for exascale applications, frameworks and runtimes

**GASNet-EX** low-level layer that provides a network-independent interface suitable for Partitioned Global Address Space (PGAS) runtime developers

**UPC++** C++ PGAS library for application, framework and library developers, a productivity layer over GASNet-EX

[https://crd.lbl.gov/pagoda](https://crd.lbl.gov/pagoda)
Some motivating applications

Many applications involve asynchronous updates to irregular data structures

- Adaptive meshes
- Sparse matrices
- Hash tables and histograms
- Graph analytics
- Dynamic work queues

Irregular and unpredictable data movement:

- **Space**: Pattern across processors
- **Time**: When data moves
- **Volume**: Size of data
Some motivating system trends

The first exascale systems will appear soon

- Cores per node is growing (including GPU cores)
- Network transfers becoming smaller and more frequent
- Network latency is **not** improving

**Need to reduce communication costs in software**

- Minimize software overhead per transfer
- Overlap communication to hide latency
- Use simpler communications protocols (RDMA)
Reducing communication overhead using one-sided RMA

Let each process directly access another’s memory via a global pointer. Communication is **one-sided**, also known as Remote Memory Access (RMA):

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

**two-sided message**

- **message id**
- **data payload**

**one-sided put message**

- **address**
- **data payload**

- All metadata provided by the initiator, rather than split between sender and receiver
- Supported in network hardware through RDMA (Remote Direct Memory Access)

Looks like shared memory: shared data structures with asynchronous access
One-sided vs two-sided message performance

Uni-directional Flood Bandwidth (many-at-a-time)

SUMMARY

(+): MPI ISend/Irecv
2-sided, while others are all 1-sided (RMA)

(○, ○): MPI RMA typically outperforms 2-sided MPI

(×, ×): GASNet-EX consistently outperforms MPI (2-sided and RMA)
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
- In conventional shared memory, pointers do not typically encode affinity
The PGAS model

Partitioned Global Address Space

- Support global memory, leveraging the network’s RDMA capability
- Distinguish private and shared memory
- Separate synchronization from data movement

Languages that provide PGAS: UPC, Titanium, Chapel, X10, Co-Array Fortran (Fortran 2008)

Libraries that provide PGAS: UPC++, OpenSHMEM, Co-Array C++, Global Arrays, DASH, MPI-RMA

This presentation is about UPC++

- C++ library implementation of the PGAS model
- Leverages productivity of C++
- Adds Remote Procedure Call (RPC) to complement RMA
- Extends global address space to encompass device memories (GPUs)
UPC++: a C++ PGAS library
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
- OpenMP and CUDA can be mixed with UPC++ as in MPI+X
Compiler-free: UPC++ global pointers

Global pointers are a C++ class 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:

```cpp
upcxx::global_ptr<double> g;
```
Low-overhead RMA: performance on Cori Haswell nodes

Round-trip Put Latency (lower is better)  
Unidirectional Flood Put Bandwidth (higher is better)

Data collected on Cori Haswell (https://doi.org/10.25344/S4V88H)
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 and scalability

- All communication is syntactically explicit
- All communication is asynchronous: futures and promises
- Scalable data structures that avoid unnecessary replication
Asynchronous communication (RMA) 1 of 2

By default, all communication operations are split-phased

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

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

A UPC++ future holds a value and a state: ready/not-ready
Wait returns the result when the rget completes
Asynchronous communication (RMA) 2 of 2

UPC++ RMA includes the following functions

- Scalar `rget` (on previous slide)
- Vector `rget` (returns an *empty* future, not holding value)
- Scalar `rput`
- Vector `rput`
- Non-contiguous RMA for scatter/gather
- Atomic memory operations

All RMA is performed using such syntactically explicit calls

- UPC++ does *not* communicate via operators (`*`, `-`, `[]`)
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, returning a future
2. Target process executes \( fn(arg1, \ arg2) \) at some later time determined at the target
3. Result becomes available to the initiator via the future

Let’s imagine that process 0 performs this RPC

```cpp
int area(int a, int b) { return a * b; }
...
... = upcxx::rpc(p, area, a, b).wait();
```

1. \( \text{upcxx}::\text{rpc}(p, \text{area}, a, b) \)
2. \( \{\text{"area"}, a, b\} \)
3. Result ready on process 0
Futures and callbacks 1 of 3

RMA and RPC both return a future object, which represents an operation that may or may not be complete.

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

```cpp
upcxx::future<int> f1 = upcxx::rpc(p, area, a, b);
// unrelated work...
... = 2 * f1.wait();
```

The `then()` method can be used to attach a callback to a future.

```cpp
upcxx::future<int> f2 =
    upcxx::rpc(p, area, a, b)
    .then([](int value) { return 2 * value; });
// unrelated work...
... = f2.wait();
```
Futures and callbacks 2 of 3

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

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

```cpp
// example code

global_ptr<int> source = ...;
global_ptr<double> target = ...;
...

future<> fut1 =
  rget(source)
  .then([](int value) { return std::log(value); })
  .then([target](double value) { return rput(value, target); });
```

Note: dropping *upcxx::* namespace in examples from this point forward.
Use of *syntax highlighting* will continue.
Futures and callbacks 3 of 3

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<> fut2 =
    when_all(rget(source_i), rget(source_d))
    .then([target](int a, double b) { return rput(a*b, target); });
```

Hargrove / UPC++ Overview / 22 Feb 2021 / upcxx.lbl.gov
Distributed objects 1 of 3

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

\[
\text{dist\_object}\langle T \rangle(\text{value}, \text{team \& team} = \text{world}());
\]

Processes share a name for the object, but each has its own local value.
Construction is collective, but not synchronizing.
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
- Can be constructed over teams (process subsets)

\[
\text{dist\_object}\langle\text{int}\rangle
\]

\[
\text{all\_nums}(\text{rand}());
\]

- Process 0: all\_nums = 3
- Process 1: all\_nums = 8
- Process p: all\_nums = 42
Distributed objects 2 of 3

As an artificial example, the following implements a naïve sum reduction:

```cpp
dist_object<double> dobj(some_computation());

// Rank 0 computes the sum:
if (rank_me() == 0) {
    double sum = *dobj;
    for (int i = 1; i < rank_n(); ++i)
        sum += dobj.fetch(i).wait();
    std::cout << "Sum is " << sum << std::endl;
}
```

Note that UPC++ does have actual split-phase reductions

More realistic use cases include a “directory” of pointers or objects
Distributed objects 3 of 3

This can easily be converted into an asynchronous sum reduction:

```cpp
dist_object<double> dobj(some_computation());
// Rank 0 computes the sum:
if (rank_me() == 0) {
    future<double> f = make_future(*dobj);
    for (int i = 1; i < rank_n(); ++i)
        f = when_all(f, dobj.fetch(i)).then([&](double a, double b){ return a+b; });
    std::cout << "Sum is " << f.wait() << std::endl;
}
```

Note that UPC++ does have actual split-phase reductions

More realistic use cases include a “directory” of pointers or objects
The shared segment

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

```cpp
global_ptr<double> grid_gptr = new_array<double>(N);
```

Allocation is **not** collective
- Each process allocates its own memory without synchronization
- Explicit synchronization may be required before use

Allocation is **not** symmetric
- Newly allocated pointers must be communicated to other processes before they can be used to access the data

Global pointers can be “downcast” to a raw C++ pointer

```cpp
double *my_grid = grid_gptr.local();
```
- Downcast operation is valid on any process in **local team**, which typically includes all processes on the same compute node
Additional features

- Memory kinds
  - Efficient RMA to and from GPU memory
- Teams and collectives
  - Barrier, reductions, etc.
- Shared memory hierarchy and `local_team`
  - Load-store access to intra-node shared memory
- Generalized completion
  - More options than just a single future per operation
- Remote atomics
  - Including offload to network hardware
- Serialization
  - Handling of non-trivial types in RPC
- Views
  - Optimized handling of collections in RPCs
- Non-contiguous RMA
  - Vector, indexed and strided
RPC and progress
RPC and progress

Review: high-level overview of an RPC's execution

1. Initiator injects the RPC to the *target* process, returning a future
2. Target process executes $fn(arg1, arg2)$ at some later time determined at the 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 run user callbacks asynchronously.

This design decision keeps the runtime lightweight and simplifies synchronization:

- **RPCs and callbacks** are run sequentially 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()`

Most applications do not require explicit calls to `progress()`

A program may invoke calls with user-level progress when it expects local callbacks and remotely invoked RPCs to execute

- Enables the user to decide when it is desirable (and correct) to do so

User-level progress executes some number of outstanding callbacks

- “Some number” could be zero, so may need to periodically invoke when expecting callbacks
- Callbacks may initiate communication
- May chain new callbacks on completion of communication (cannot wait)
Threads, personas and LPC

UPC++ interoperates with diverse threading models

**persona**: abstraction of thread-specific internal state

- *Default* persona created implicitly for each thread
- Additional personas can be “owned” by different threads at different times

*Master* persona: a non-default persona created implicitly and initially owned by the thread invoking `init()`

- Incoming RPC’s only processed during user-level progress by the thread owning the master persona
- One can manage the master persona to construct an async progress thread

Local procedure calls (LPC): allows any thread to enqueue work for any persona

```
future<T> persona::lpc(Callable)
```
Using UPC++
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 backend C++ compiler with the appropriate arguments (\(-I/-L\) etc).
- We also provide other mechanisms for compiling without the wrapper
  - upcxx-meta
  - CMake package

Launch wrapper:

$ upcxx-run -np 4 ./hello-world.exe

- Arguments similar to other familiar tools
- Also support use of platform-specific tools, such as \(srun\), \(jsrun\) and \(aprun\).
Using UPC++ at US DOE Office of Science centers

NERSC's Cori

```
$ module load upcxx
```

ALCF's Theta

```
$ module use /projects/CSC250STPM17/modulefiles
$ module load upcxx
```

OLCF's Summit

```
$ module use $WORLDWORK/csc296/summit/modulefiles
$ module load upcxx
```

More details, including compile and run examples for all three centers, are available in documentation linked from upcxx.lbl.gov/training
Application case studies
Several applications have been written using UPC++, resulting in improved programmer productivity and runtime performance. Examples include:

- MetaHipMer, a genome assembler (subject of Feb 8 seminar)
- symPack, a sparse symmetric matrix solver
- Pond, an actor-based shallow water simulation
- Sim-COV, agent-base simulation of lungs with COVID
symPack

A sparse symmetric matrix solver
symPACK: a solver for sparse symmetric matrices

1) Data is produced
2) Notifications sent using **rpc_ff** †
   - Enqueues a **global_ptr** to the data
   - Manages dependency count
3) When all data is available, task is moved in the “data available tasks” list
4) Data is retrieved using **rget**
   - Once transfer is complete, update dependency count
5) When everything has been transferred, task is moved to the “ready tasks” list

† **rpc_ff** is “RPC Fire and Forget” – an RPC which does not return an acknowledgement to the initiating process.

[Link to symPACK](https://upcxx.lbl.gov/sympack)

---

Work and results by Mathias Jacquelin, funded by SciDAC CompCat and FASTMath

Hargrove / UPC++ Overview / 22 Feb 2021 / upcxx.lbl.gov
symPACK a solver for sparse symmetric matrices

Matrix is distributed by supernodes
• 1D distribution
  • Balances flops, memory
  • Lacks strong scalability
• New 2D distribution (to appear)
  • Explicit load balancing, not regular block cyclic mapping
  • Balances flops, memory
  • Finer granularity task graph

Strong scalability on Cori Haswell:
• Up to 3x speedup for Serena
• Up to 2.5x speedup for DG_Phosphorene_14000

UPC++ enables the finer granularity task graph to be fully exploited
• Better strong scalability
symPACK strong scaling experiment

Run times for Flan_1565

Max speedup: 1.85x

Experiment done on NERSC Cori KNL Cray XC Aries

Work and results by Mathias Jacquelin, funded by SciDAC CompCat and FASTMath

Processes

N=1,564,794  nnz(L)=1,574,541,576
symPACK strong scaling experiment

Run times for audikw_1

Max speedup: 2.13x

Experiment done on NERSC Cori Haswell Cray XC Aries

Work and results by Mathias Jacquelin, funded by SciDAC CompCat and FASTMath

N=943,695  nnz(L)=1,261,342,196
UPC++ provides productivity + performance for symPack

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 and futures
• Outperform state-of-the-art sparse symmetric solvers

https://upcxx.lbl.gov/sympack
Pond

An actor-based shallow water simulation
A UPC++ actor library and its evaluation on a shallow water application

Alexander Pöppl, Scott B. Baden, Michael Bader

In 2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI+X (PAW-ATM), November 2019.

https://doi.org/10.25344/S43G60
UPC++ actor library

- Actors
  - Encapsulate functionality, data and behaviors
  - Behaviors via finite state machines
  - Data is local to each actor
  - Actors communicate via Ports
  - Triggered to compute when data arrives via connected Ports
- The application developer
  - Subclasses and provides the `act()` method
  - Controls which Ports are connected
  - Uses ports for communication among actors
- Implementation uses UPC++ RPC and LPC
- Multiple execution strategies:
  - Rank-based and Task-based
Pond – A shallow water proxy application

- Solves the shallow water Navier-Stokes equations
  - Applicable to tsunami modeling
- Based on prior applications
  - SWE, a BSP-based code written using MPI and OpenMP
  - SWE-X10, an actor-based X10 application written using the actorX10 library
Weak scaling comparison on Cori KNL

Performance of Pond (UPC++) compared to SWE (MPI + OpenMP) and SWE-X10 (X10)

Radial Dam Break Scenario

4096² grid cells per node

Both UPC++ versions exceed the performance of SWE (MPI), with speedups shown in the bar graph.
Sim-COV

Agent-base simulation of lungs with COVID
SIM-Cov: spatial model of immune response to viral lung infection

M. Moses, J. Cannon (UNM), S. Forrest (ASU) and S. Hofmeyr (LBNL)

- The immune response to SARS-Cov-2 plays a critical role in determining the outcome of Covid-19 in an individual
- Most of what you hear about the immune response is focused on antibodies
- However, antibodies can only stop a virus that is outside a host cell
- Once it has invaded a cell, it is the "job" of the T cells to attack the virus
- Understanding how T cells detect and clear the virus is fundamental to understanding disease progression and resolution

To investigate this, the team is building a 3D agent-based model of the lungs, called SIM-Cov
SIM-Cov implementation

- Goal is to 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

- SIM-Cov 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
SIM-Cov components
Speculative simulations to explore role of T cells in disease severity

Mild infection:
- high T cell response
- controls viral infection
- recovery by day 10 (viral drops near zero)

Severe infection:
- low T cell response
- fails to control infection
- initial drop in viral load but surge later on
- corresponds to a common progression actually seen in severe disease (people feel better then get a lot worse)
Visualization of Prototype Simulation

- Run headless and visualize afterwards using Paraview
- Spread of infection from single focal point
- Very small 2D area without branching structures
THANK YOU
Acknowledgements

In addition to those credited in the Application Case Studies, this presentation includes the efforts of the following past and present members of the Pagoda group and collaborators:

John Bachan, Scott B. Baden, Dan Bonachea, Rob Egan, Max Grossman, Paul H. Hargrove, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, Daniel Waters, Katherine Yelick

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.
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

“UPC++ has an excellent blend of ease-of-use combined with high performance. Features such as RPCs make it really easy to rapidly prototype applications, and still have decent performance. Other features (such as one-sided RMAs and asynchrony) enable fine-tuning to get really great performance.”

-- Steven Hofmeyr, LBNL

“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