

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

Amir Kamil

<https://go.lbl.gov/CUF23>
pagoda@lbl.gov



Applied Mathematics and Computational Research Division
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:

Hadia Ahmed, John Bachan, Scott B. Baden, Dan Bonachea, Johnny Corbino, Rob Egan, Max Grossman, Paul H. Hargrove, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Colin MacLean, Damian Rouson, Erich Strohmaier, 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.

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

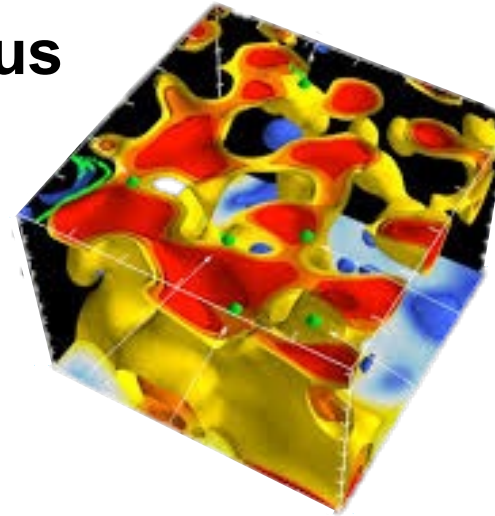
Design principles for performance

- All communication is syntactically explicit
- All communication is asynchronous: futures and promises
- Scalable data structures that avoid unnecessary replication

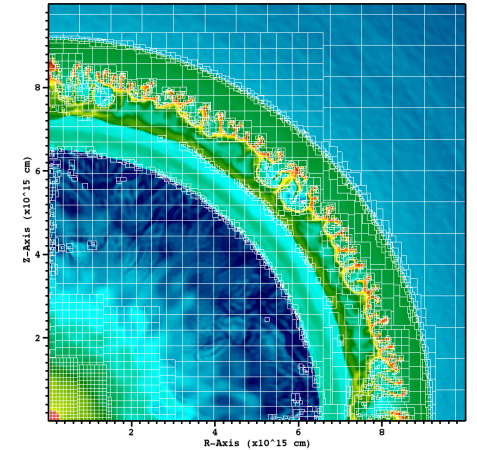
Some motivating applications

Many applications involve asynchronous updates to irregular data structures

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



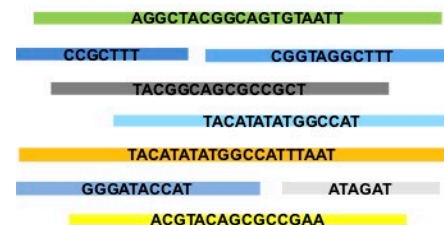
Seismo, Berkeley



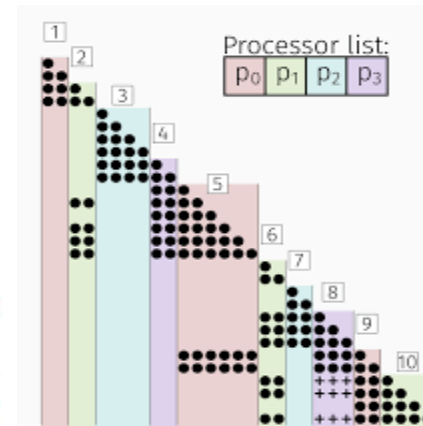
AMReX

Irregular and unpredictable data movement:

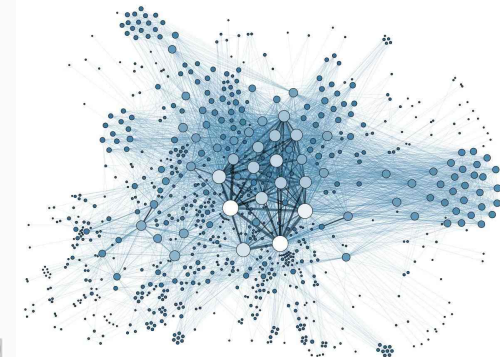
- *Space*: Pattern across processors
- *Time*: When data moves
- *Volume*: Size of data



ExaBiome



SymPACK



Graph analytics

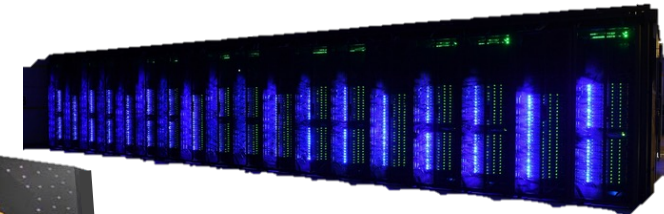
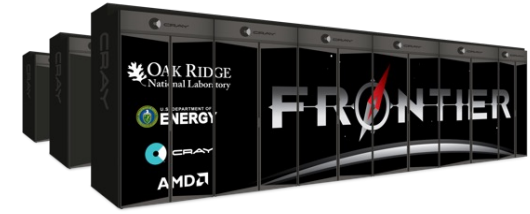
Some motivating system trends

The first exascale systems appeared in 2022

- Cores per node is growing
- Accelerators (e.g. GPUs) are becoming more important
- Latency is not improving

Need to reduce communication costs in software

- Overlap communication to hide latency
- Reduce memory using smaller, more frequent messages
- Minimize software overhead
- Use simple messaging protocols (RDMA)



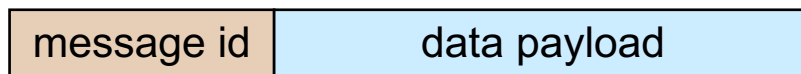
Reducing communication overhead

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

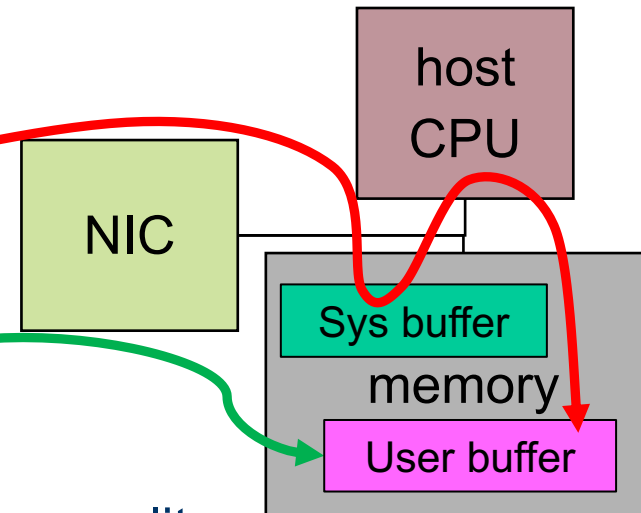
Communication is **one-sided** – there is no “receive” operation

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

two-sided message



one-sided RMA put



- 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

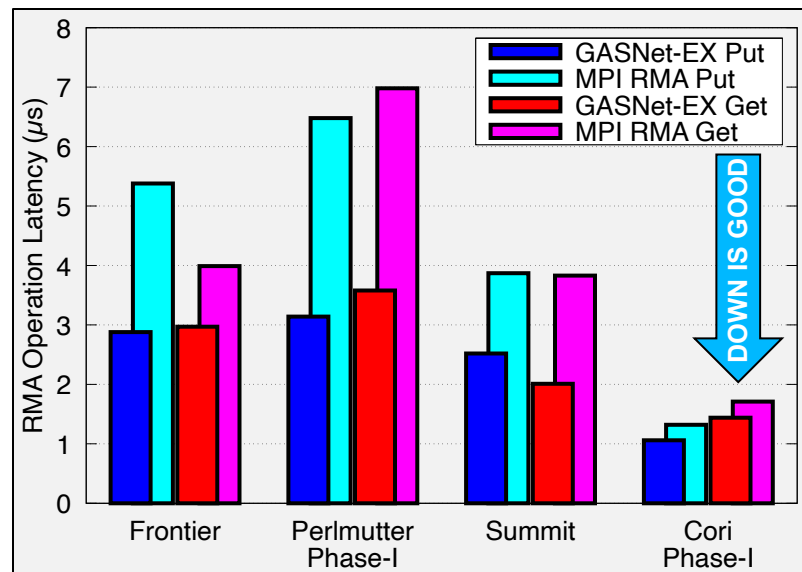
One-sided GASNet-EX vs one- and two-sided MPI

Four distinct network hardware types

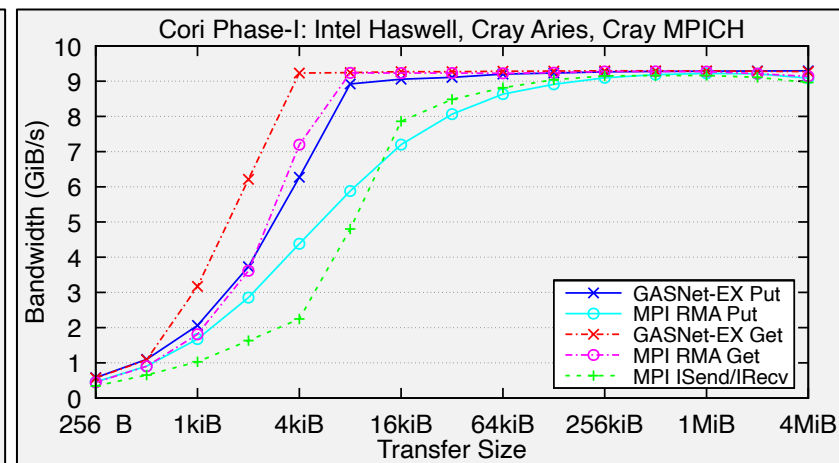
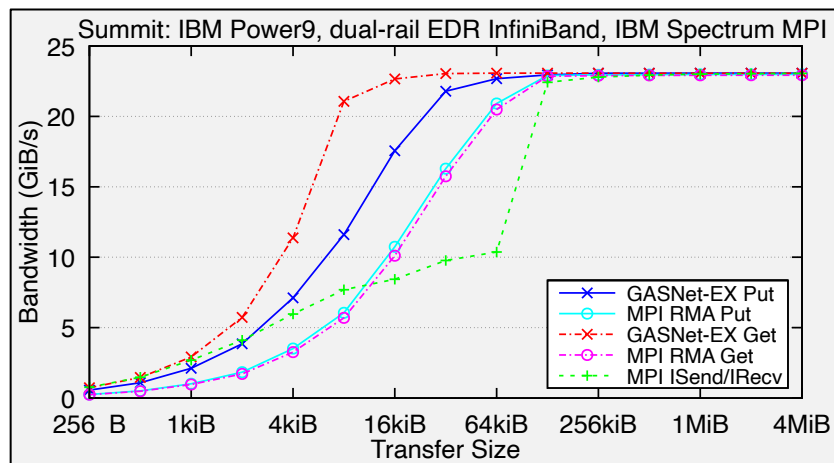
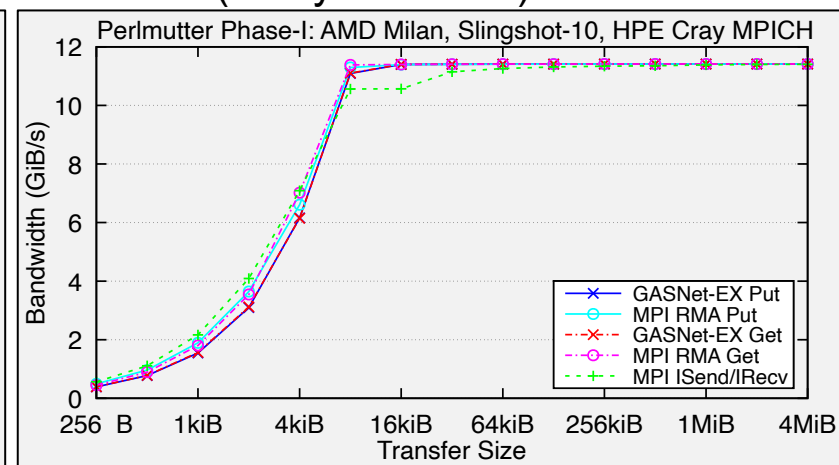
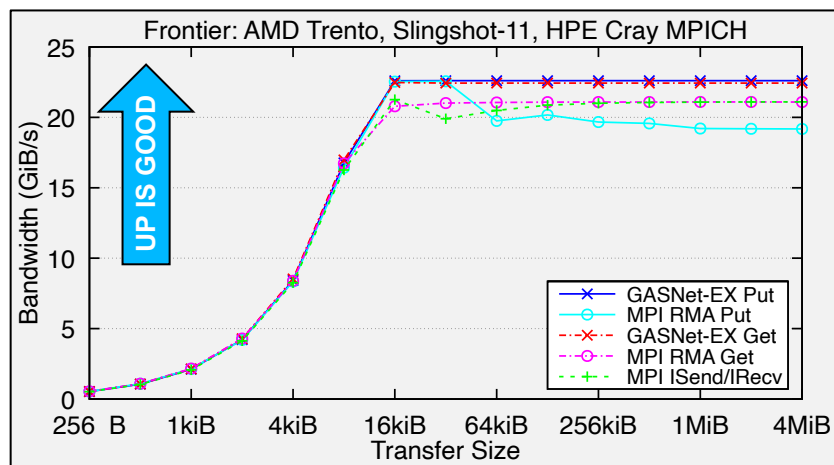
The performance of one-sided GASNet-EX matches or exceeds that of MPI RMA and message-passing:

- 8-byte Put latency 19 - 52% better
- 8-byte Get latency 16 - 49% better
- Better flood bandwidth efficiency: often reaching same or better peak at $\frac{1}{2}$ or $\frac{1}{4}$ the transfer size

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



Uni-directional Flood Bandwidth (many-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

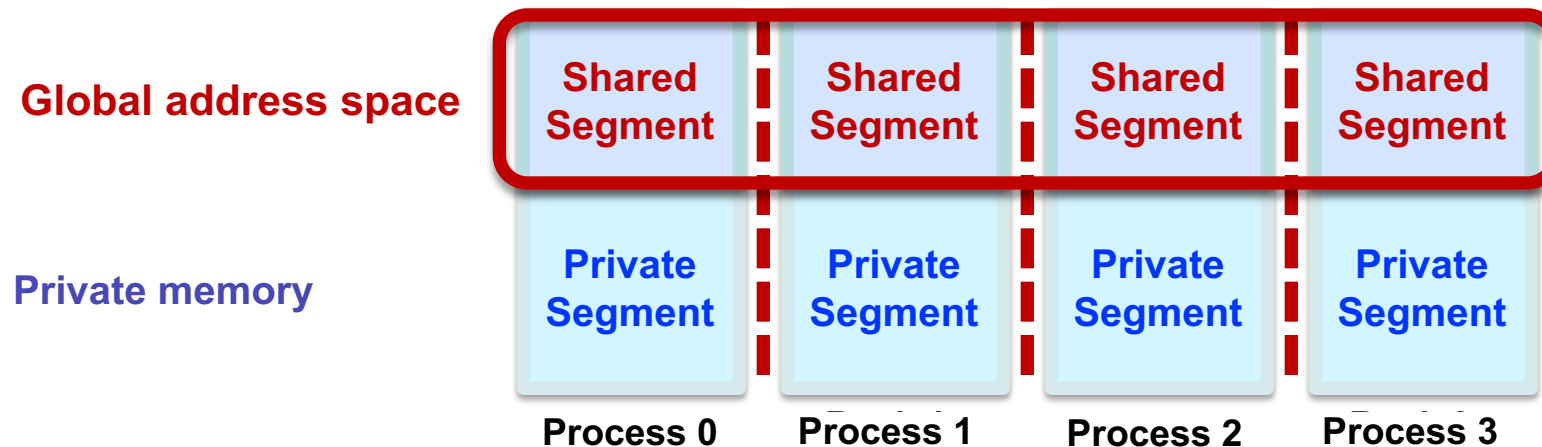
A Partitioned Global Address Space programming model

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 encode affinity



The PGAS model

Partitioned **G**lobal **A**ddress **S**pace

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

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

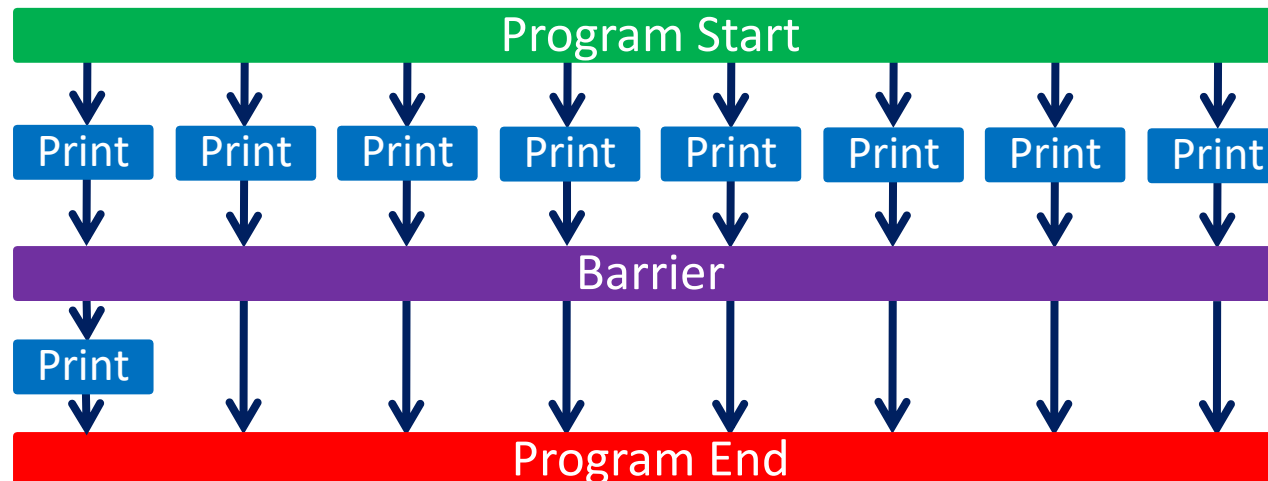
Libraries that provide PGAS: OpenSHMEM, Co-Array C++, Global Arrays, DASH, MPI-RMA

This presentation is about UPC++, a C++ library developed at Lawrence Berkeley National Laboratory

Execution model: SPMD

Like MPI and Coarray Fortran, UPC++ uses a SPMD model of execution, where a fixed number of processes run the same program

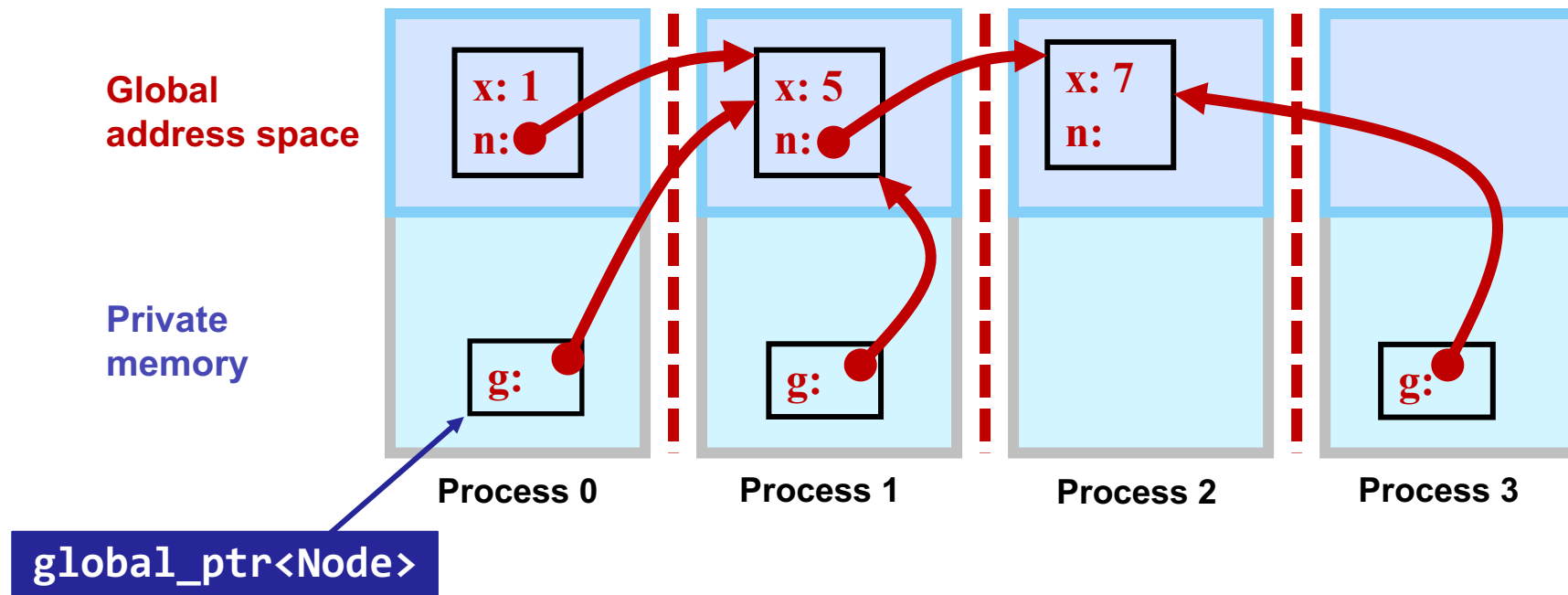
```
int main() {  
    upcxx::init();  
    cout << "Hello from " << upcxx::rank_me() << endl;  
    upcxx::barrier();  
    if (upcxx::rank_me() == 0) cout << "Done." << endl;  
    upcxx::finalize();  
}
```



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>`

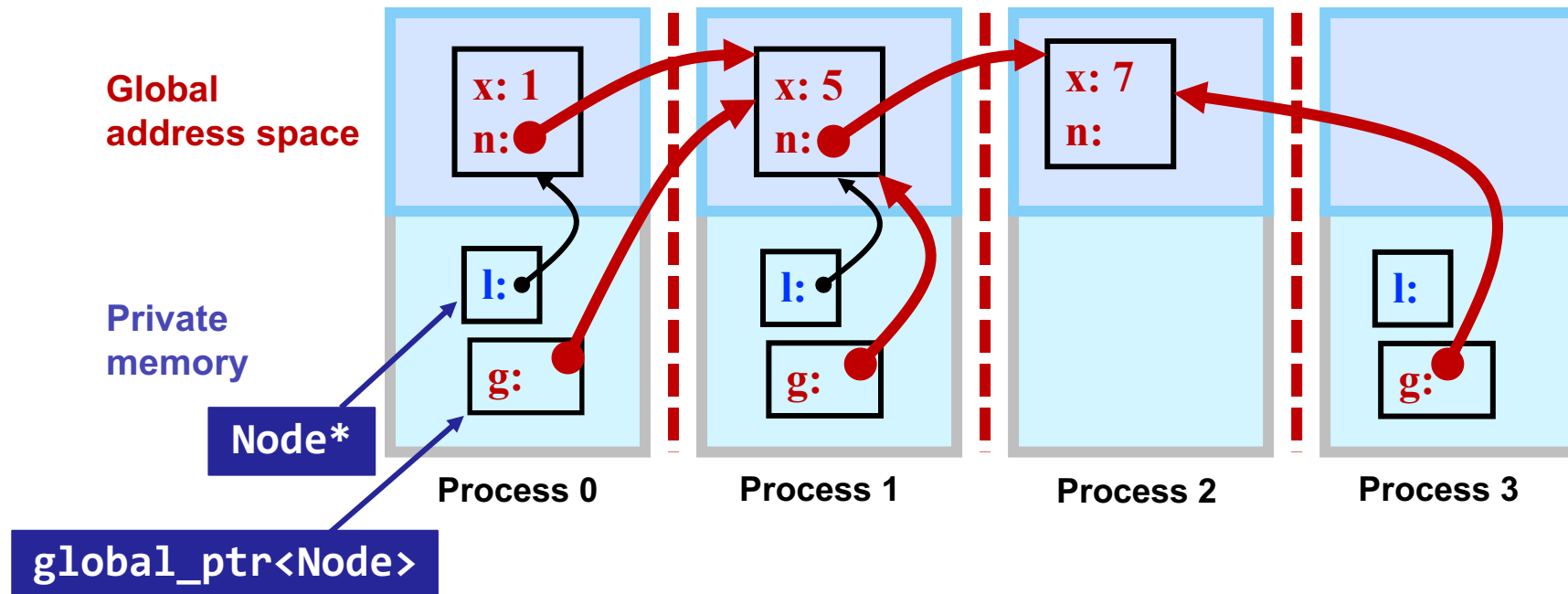


Global vs raw pointers and affinity

The affinity identifies the process that created the object

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

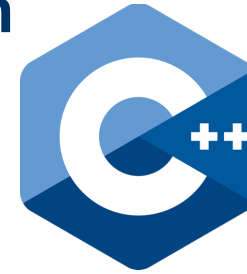
Raw C++ pointers (e.g. `Node*`) can be used on a process to refer to objects in the global address space that have affinity to that process



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)

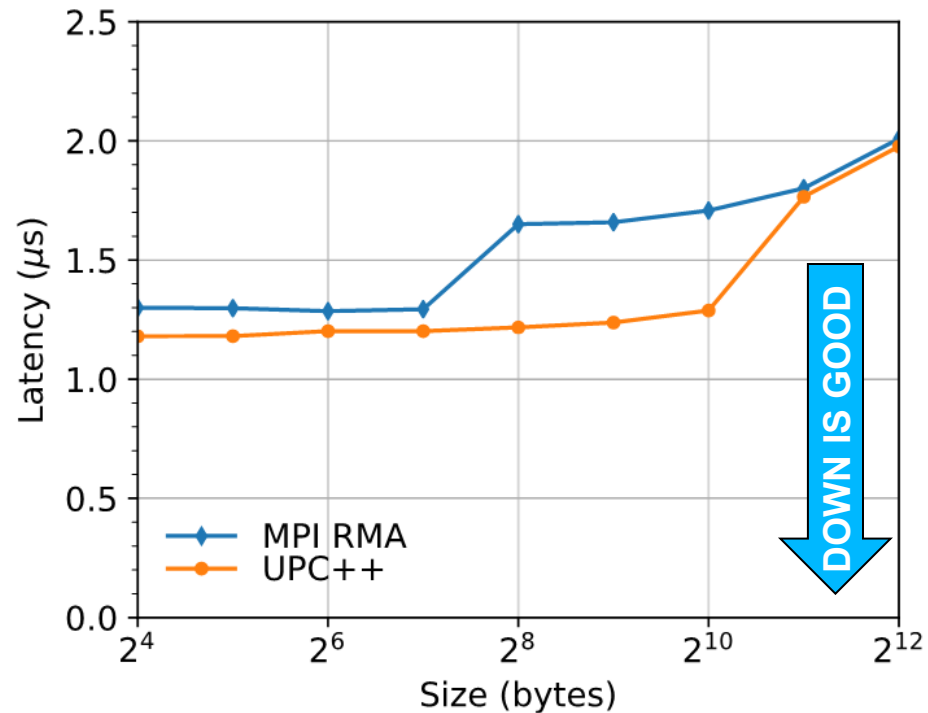
Designed for interoperability

- Same 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

UPC++ on top of GASNet

Experiments on NERSC Cori:

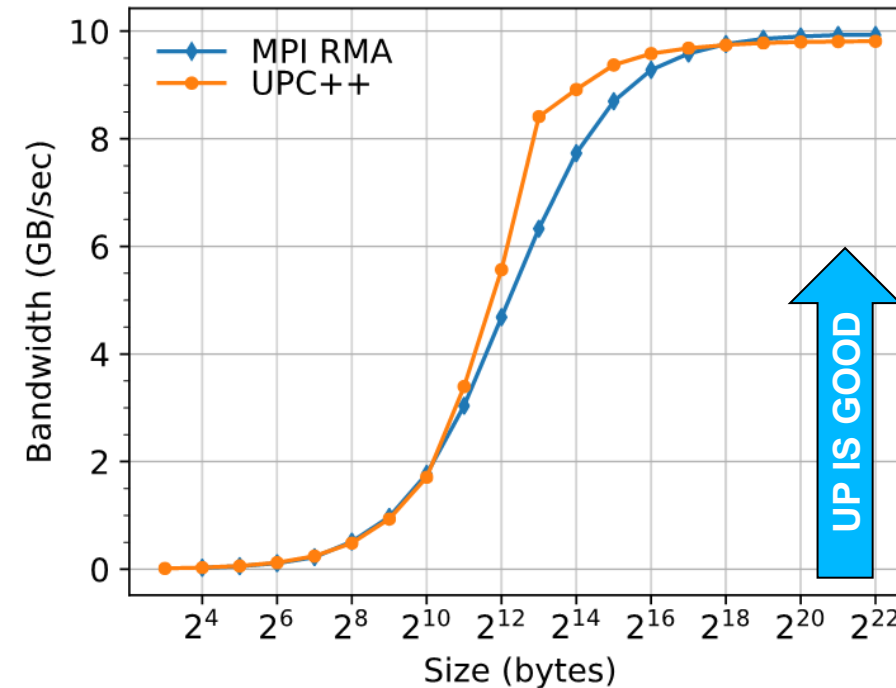
- Cray XC40 system



Round-trip Put Latency (lower is better)

Two processor partitions:

- Intel Haswell (2 x 16 cores per node)
- Intel KNL (1 x 68 cores per node)



Flood Put Bandwidth (higher is better)

Data collected on Cori Haswell (<https://doi.org/10.25344/S4V88H>)

Asynchronous communication (RMA)

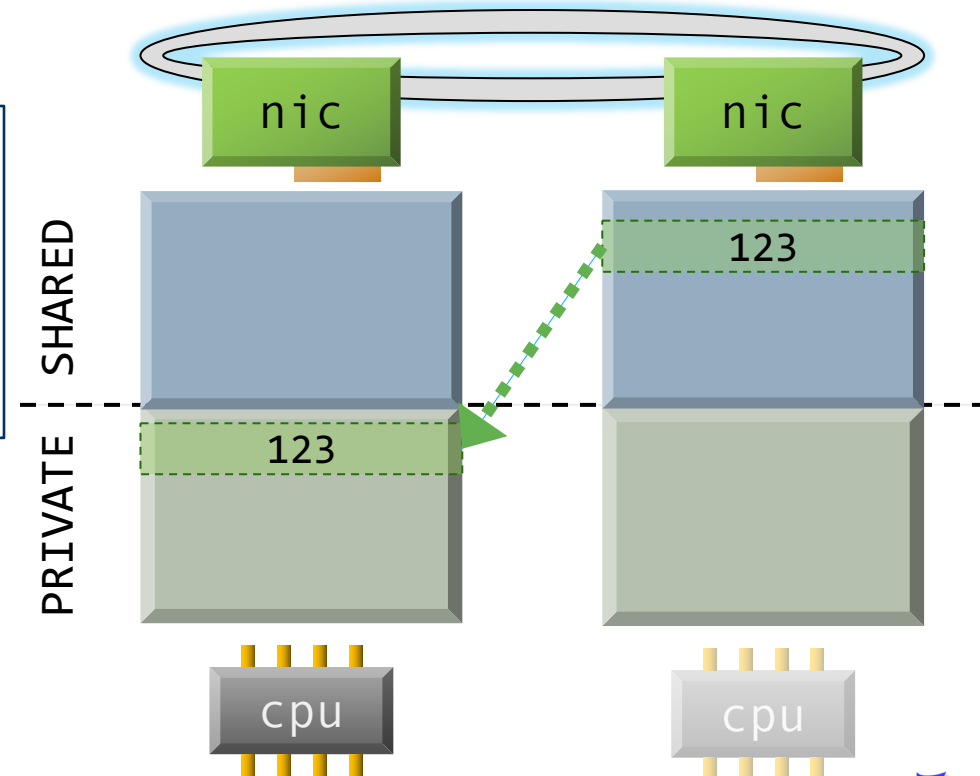
By default, all communication operations are split-phased

- **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

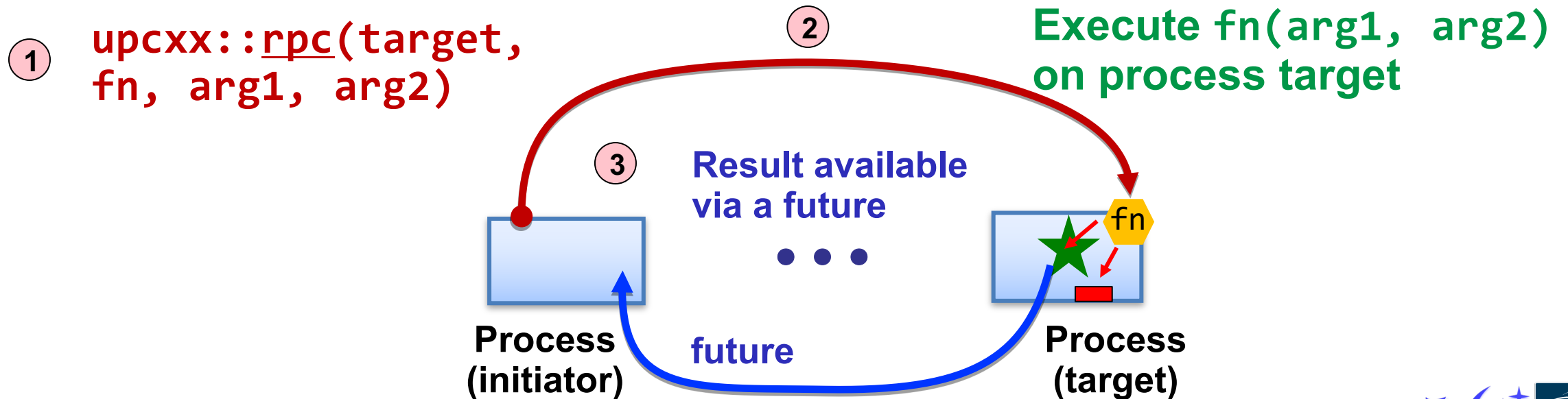


Remote procedure call (RPC)

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(arg1, 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



Hands-on: 2D heat diffusion

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

Everything needed for the hands-on activities is at:

<https://go.lbl.gov/CUF23>

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 cuf23/upcxx directory:

```
$ make run-heat2d
```

Command to run
in the terminal

```
upcxx heat2d.cpp -Wall -o heat2d
```

```
upcxx-run -N 1 -n 4 ./heat2d
```

Copy this and add arguments to change the
problem size, e.g.:

```
upcxx-run -N 1 -n 4 ./heat2d 8192 8192
```

```
[2] My Neighbors: (1, 3) My Domain: (2048,3072)
[3] My Neighbors: (2, -1) My Domain: (3072,4096)
[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
```

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

Amir Kamil

<https://go.lbl.gov/CUF23>
pagoda@lbl.gov



Applied Mathematics and Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California, USA



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

Review: Asynchronous communication (RMA)

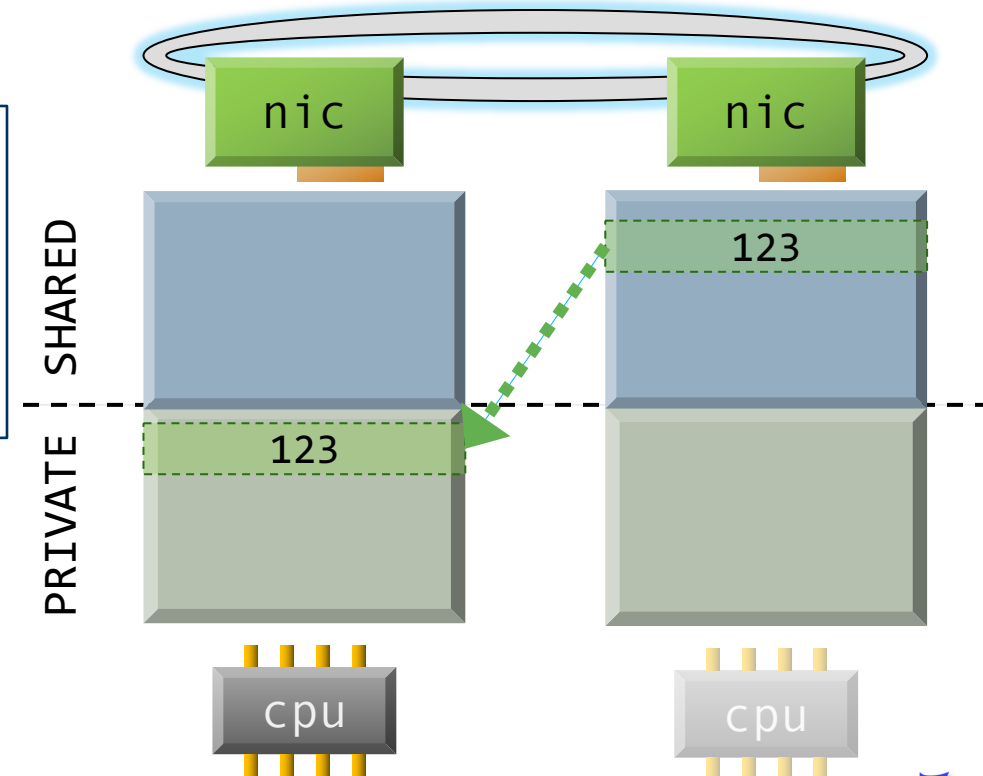
By default, all communication operations are split-phased

- **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

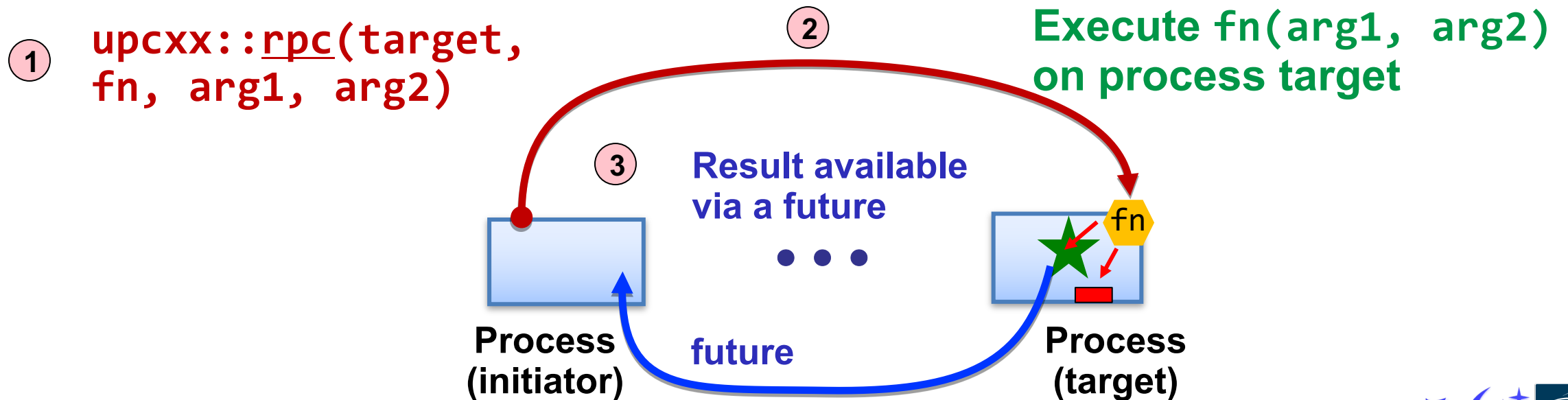


Review: Remote procedure call (RPC)

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(arg1, 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



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`.

Using UPC++ at US DOE Office of Science Centers

UPC++ installations available at ALCF (Polaris, Theta, Sunspot), NERSC (Perlmutter), and OLCF (Summit, Frontier, Crusher)

Info and examples for all three centers are available from
<https://upcxx.lbl.gov/site>

Also contains links to UPC++ source and build instructions

UPC++ works on laptops, workstations, and clusters too

Instructions for the hands-on activities in this tutorial:
<https://go.lbl.gov/CUF23>

Hands-on: Hello world compile and run

Everything needed for the hands-on activities is at:

<https://go.lbl.gov/CUF23>

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 cuf23/upcxx directory:

```
$ make run-hello-world
```

Command to run
in the terminal

```
upcxx hello-world.cpp -Wall -o hello-world
```

```
upcxx-run -N 1 -n 4 ./hello-world
```

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

```
upcxx-run -N 1 -n 8 ./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
```

Example: Hello world

```
#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();
}
```

Set up UPC++
runtime

Close down
UPC++ runtime

```
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

```
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

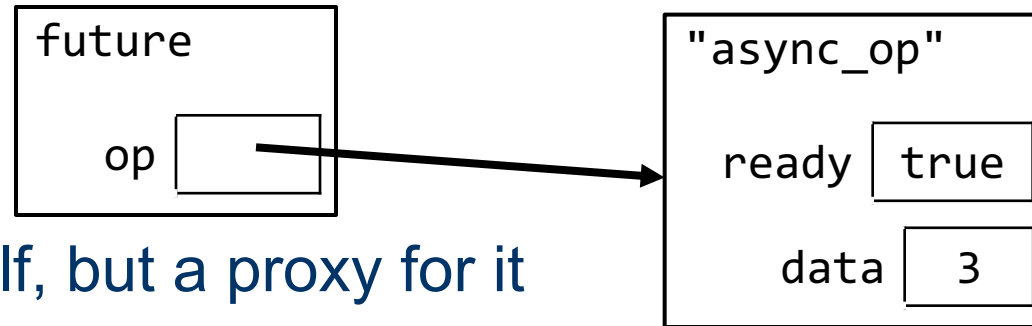
```
upcxx::future<> fut =  
    upcxx::rpc(0, [](int rank) {  
        cout << "Hello from process " << rank << endl;  
    }, upcxx::rank_me());  
  
fut.wait();
```

Empty future type that
does not hold a value,
but still tracks readiness

What is a 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

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

The then() method can be used instead to attach a callback to the future

Overlapping communication

Rather than waiting on each RPC to complete, we can launch every RPC and then wait for each to complete

```
vector<upcxx::future<int>> results;
for (int i = 0; i < upcxx::rank_n(); ++i) {
    upcxx::future<int> fut = upcxx::rpc(i, []() {
        return upcxx::rank_me();
    }));
    results.push_back(fut);
}

for (auto fut : results) {
    cout << fut.wait() << endl;
}
```

We'll see better ways to wait on groups of asynchronous operations later

1D 3-point Jacobi in UPC++

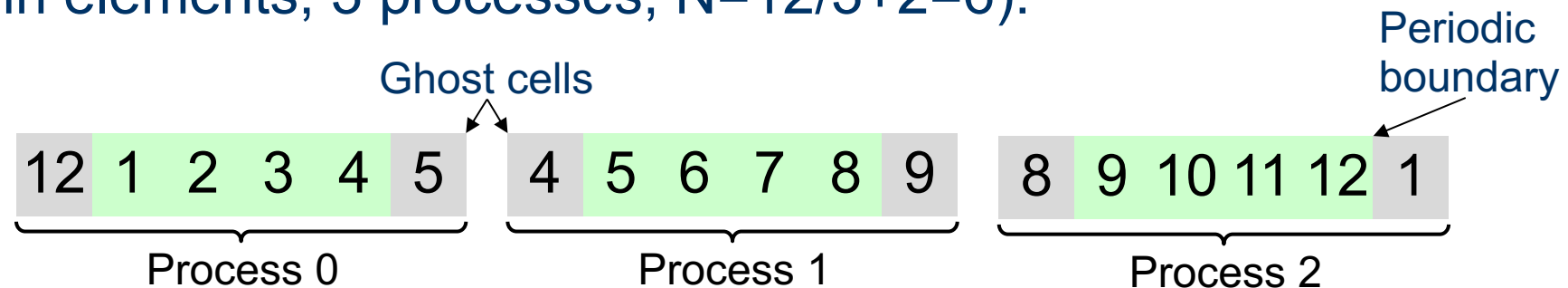
Iterative algorithm that updates each grid cell as a function of its old value and those of its immediate neighbors

Out-of-place computation requires two grids

```
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]);
```

Local grid size

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

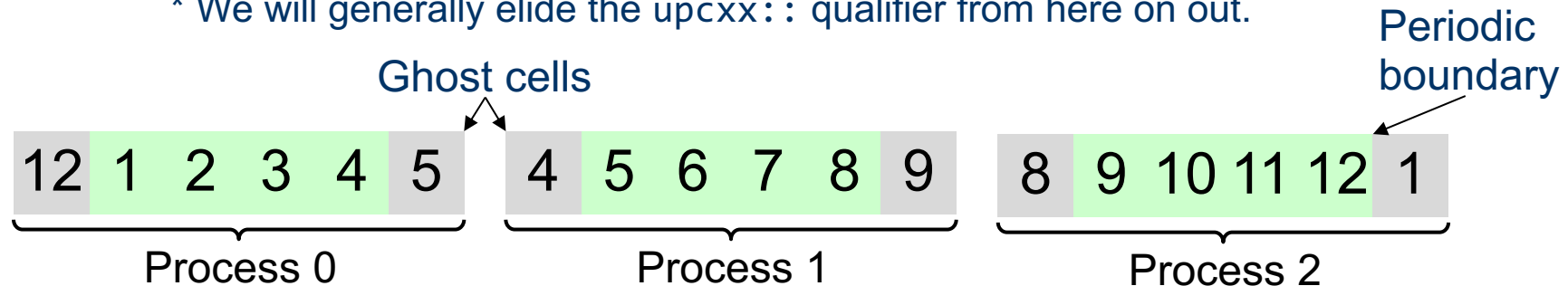
```
double *old_grid, *new_grid;
```

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

...

```
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

```
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]);
```

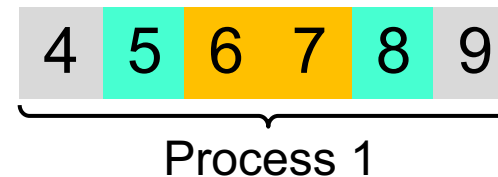
```
new_grid[1] = 0.25 *  
    (left_ghost.wait() + 2*old_grid[1] + old_grid[2]);  
new_grid[N-2] = 0.25 *  
    (old_grid[N-3] + 2*old_grid[N-2] + right_ghost.wait());
```

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

Initiate
communication

Do interior
computation

Wait for
communication
to complete and
do boundary
computation

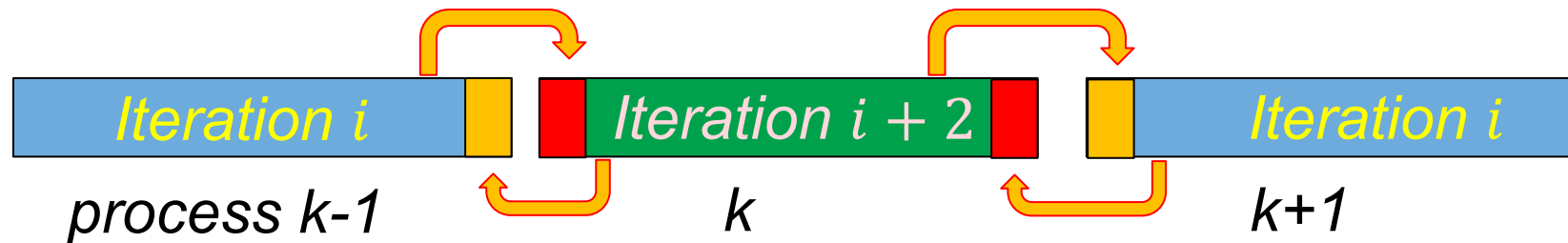


Race conditions

Since processes are unsynchronized, it is possible that a process can move on to later iterations while its neighbors are still on previous ones

- One-sided communication decouples data movement from synchronization for better performance

A *straggler* in iteration i could obtain data from a neighbor that is computing iteration $i + 2$, resulting in incorrect values



This behavior is unpredictable and may not be observed in testing

Naïve solution: barriers

Barriers at the end of each iteration provide sufficient synchronization

```
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[1] = 0.25 *  
    (left_ghost.wait() + 2*old_grid[1] + old_grid[2]);  
new_grid[N-2] = 0.25 *  
    (old_grid[N-3] + 2*old_grid[N-2] + right_ghost.wait());  
  
barrier();  
std::swap(old_grid, new_grid);  
barrier();
```

Barriers around the swap
ensure that incoming RPCs in
both this iteration and the next
one use the correct grids

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:

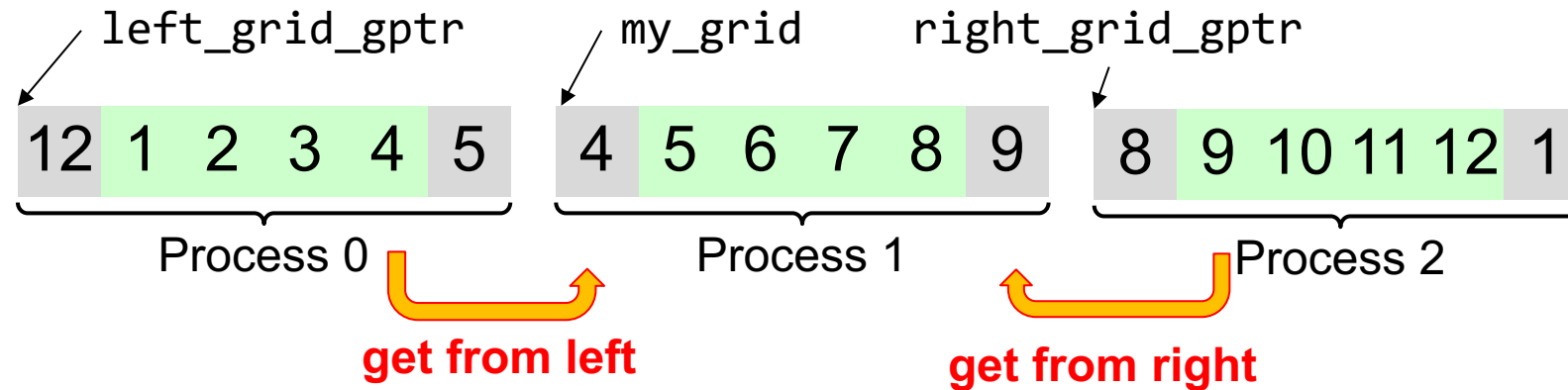
```
global_ptr<int> remote = /* ... */;  
future<int> fut1 = rget(remote);  
int result = fut1.wait();  
future<> fut2 = rput(42, remote);  
fut2.wait();
```

- Vector put and get:

```
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:

```
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

```
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()

```
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 `rget()`

```
                                Remote source (global_ptr)  Local dest ptr
                                {                          {
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)
    /* ... */;
```

Overlapped computation
on interior cells

Begin asynchronous
RMA gets

```
left_get.wait();
new_grid[1] = 0.25*(old_grid[0] + 2*old_grid[1] + old_grid[2]);

right_get.wait();
new_grid[N-2] = 0.25*(old_grid[N-3] + 2*old_grid[N-2] + old_grid[N-1]);
```

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]);  
  });
```

← Vector get does not produce a value

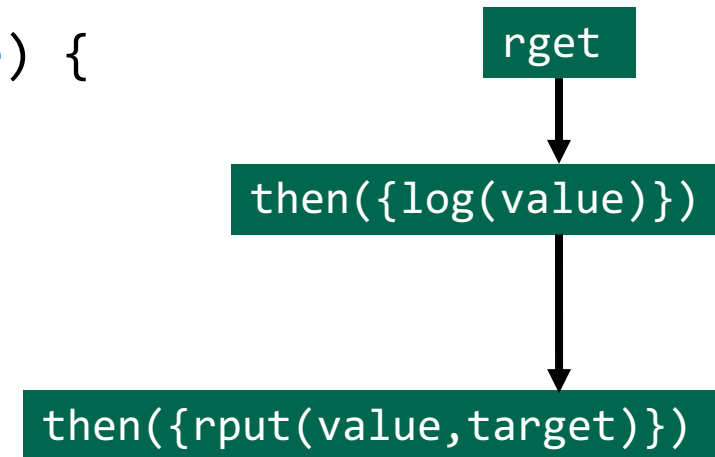
```
future<> right_update =  
  rget(right_old_grid + N - 2)  
  .then([](double value) {  
    new_grid[N-2] = 0.25 *  
      (old_grid[N-3] + 2*old_grid[N-2] + value);  
  });
```

← Scalar get produces a value

Chaining callbacks

Callbacks can be chained through calls to then()

```
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();
```



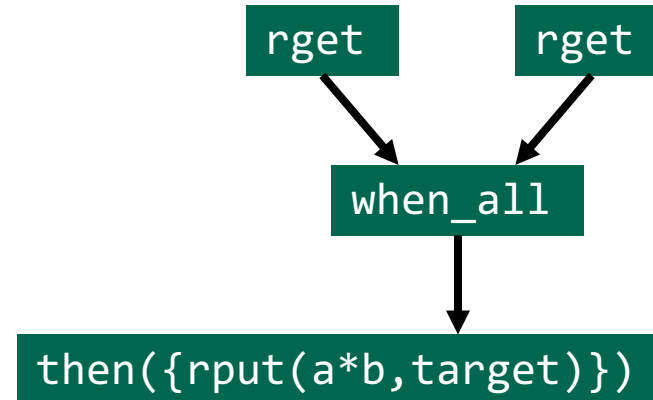
This code retrieves an integer from a remote location, computes its log, and then sends it to a different remote location

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

```
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();
```



Jacobi RMA with puts and conjoining

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

```
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

```
new_grid[1] = 0.25 * (old_grid[0] + 2*old_grid[1] + old_grid[2]);  
new_grid[N-2] = 0.25 * (old_grid[N-3] + 2*old_grid[N-2] + old_grid[N-1]);
```

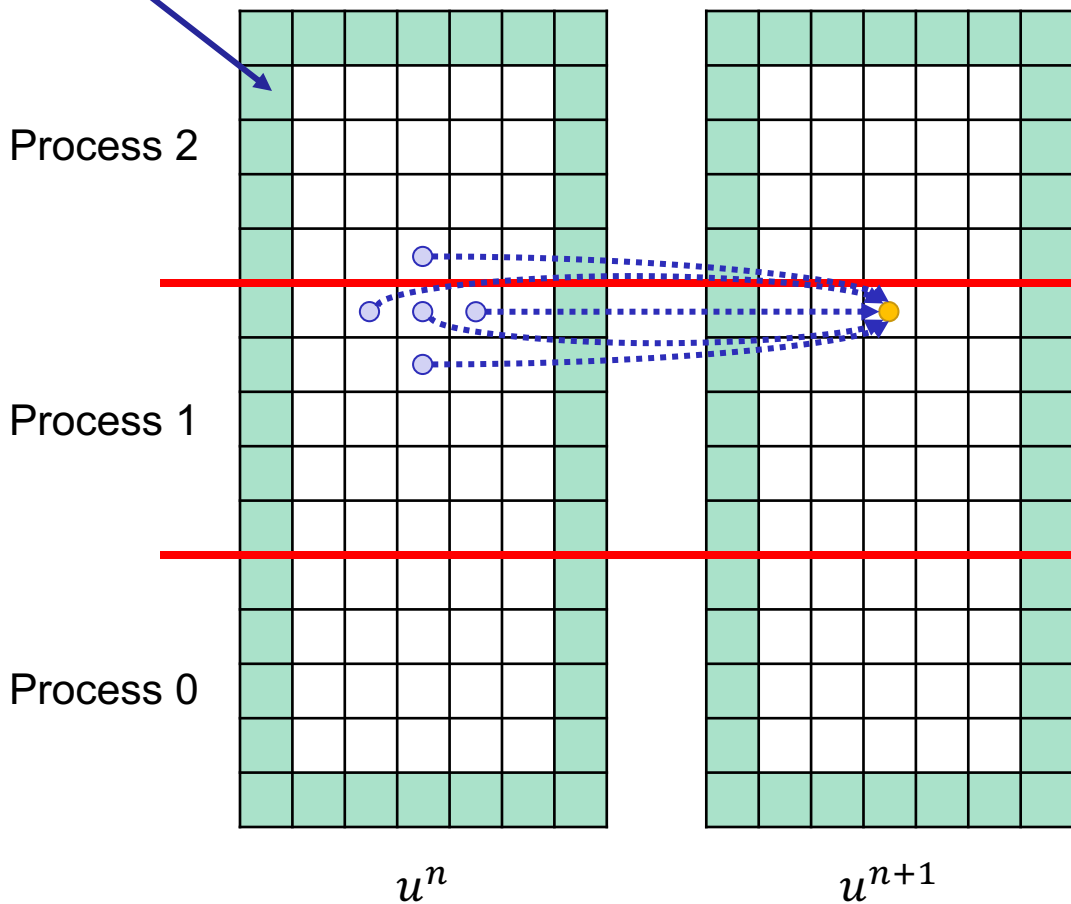
2D heat diffusion data layout

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

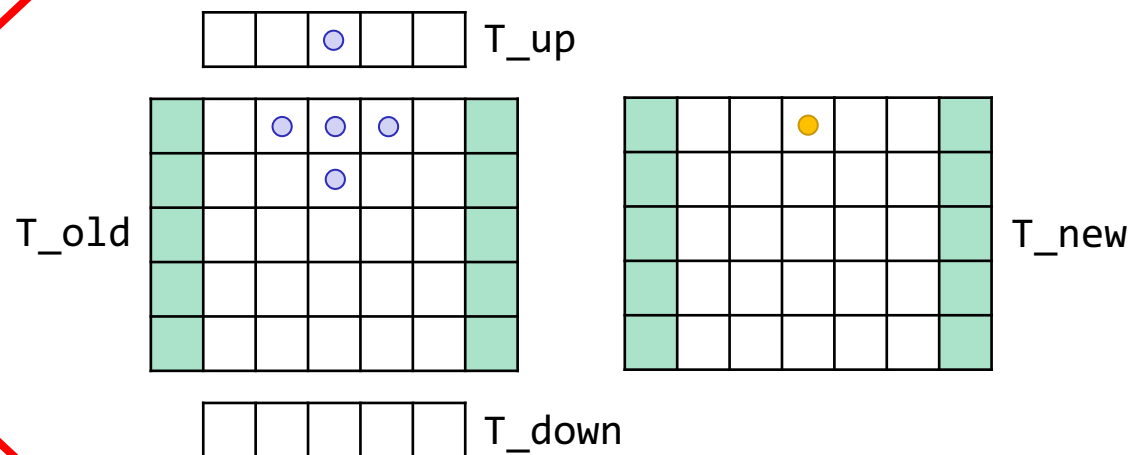
```
make run-heat2d
```

Fixed
boundary
values

Global (Abstract) View



Local (Concrete) View



“Landing zone” for
receiving data from
downward neighbor

2D heat diffusion computation

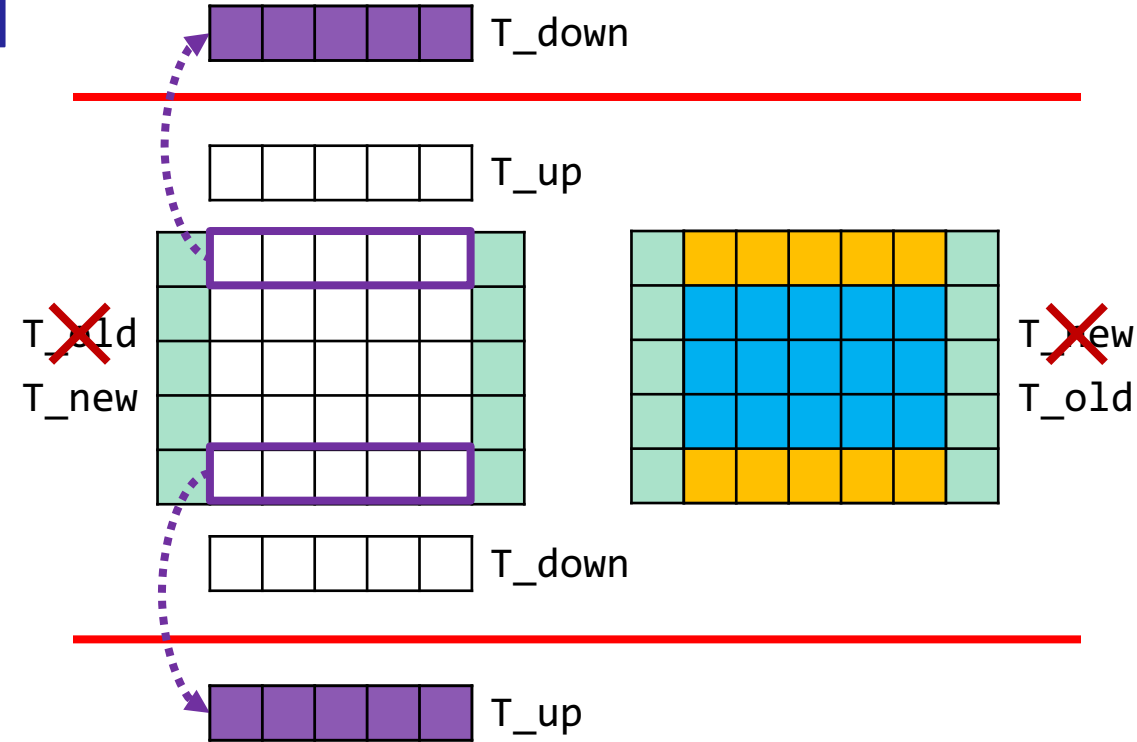
$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha(u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n + u_{i,j+1}^n + u_{i,j-1}^n)$$

make run-heat2d

Computation loop:

Global pointer to
neighbor's landing zone

```
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();
}
```



Distributed objects

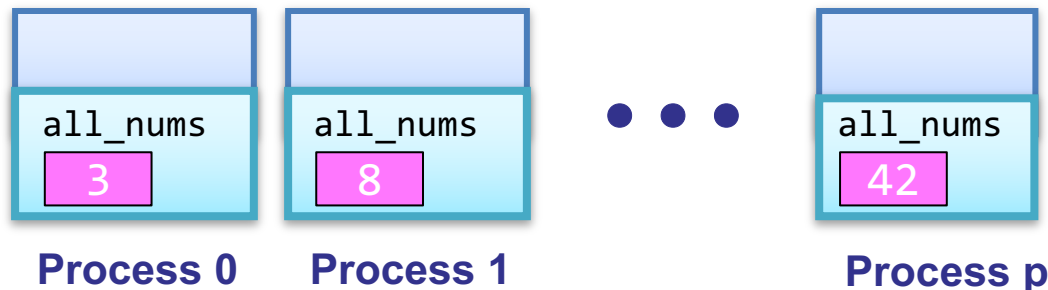
A *distributed object* is an object that is partitioned over a set of processes

```
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



```
dist_object<int>  
all_nums(rand());
```

Distributed objects in 2D heat diffusion

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

```
global_ptr<double> down_in, up_in;
```

```
if (lo != 0) {
```

```
    down_in = new_array<double>(X);
```

```
    T_down = down_in.local();
```

```
}
```

```
if (hi != Y) {
```

```
    up_in = new_array<double>(X);
```

```
    T_up = up_in.local();
```

```
}
```

```
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

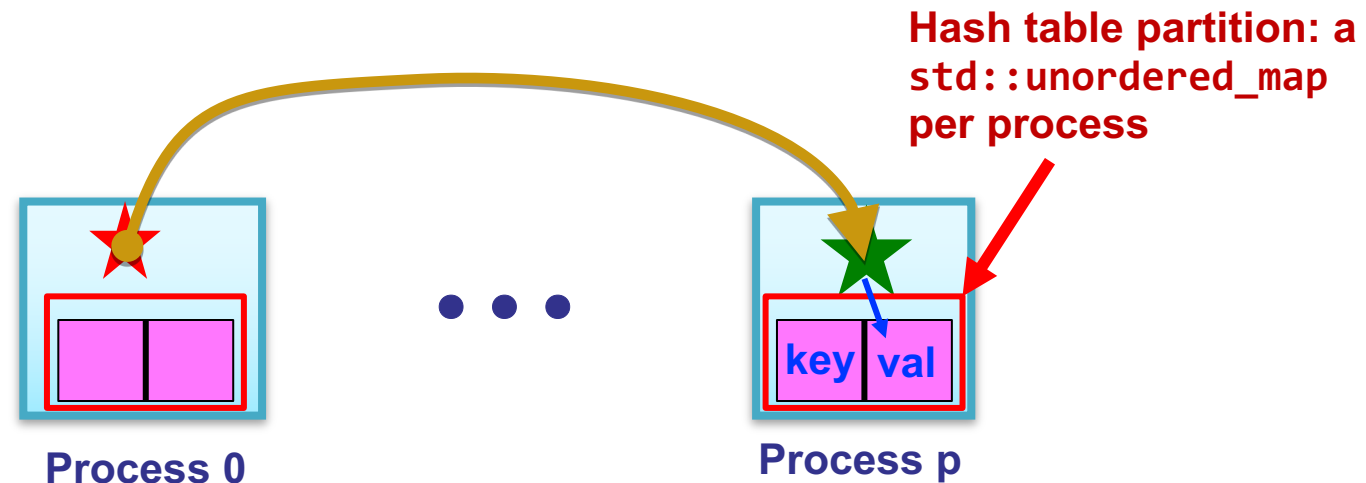
Fetch landing-zone pointer from the neighbor below

Ensure that all fetches have completed before the distributed objects are destroyed

Hands-on: Distributed hash table (DHT)

Distributed analog of `std::unordered_map` (similar to Python dict, Java HashMap)

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



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>>;
```

Define an abbreviation for a helper type

```
// Construct empty map
```

```
dobj_map_t local_map{{}};
```

Computes owner for the given key

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

DHT insertion

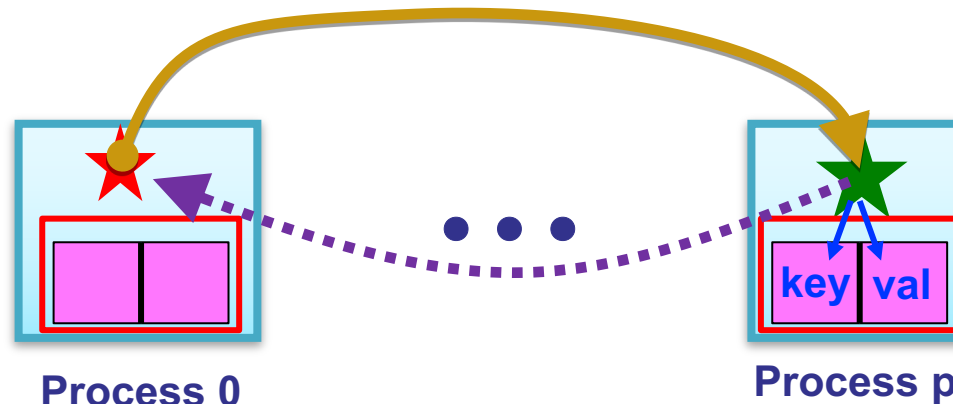
Insertion initiates an RPC to the owner and returns a future that represents completion of the insert

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

Send RPC to the process
determined by key hash

Key and value passed
as arguments to the
remote function

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

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

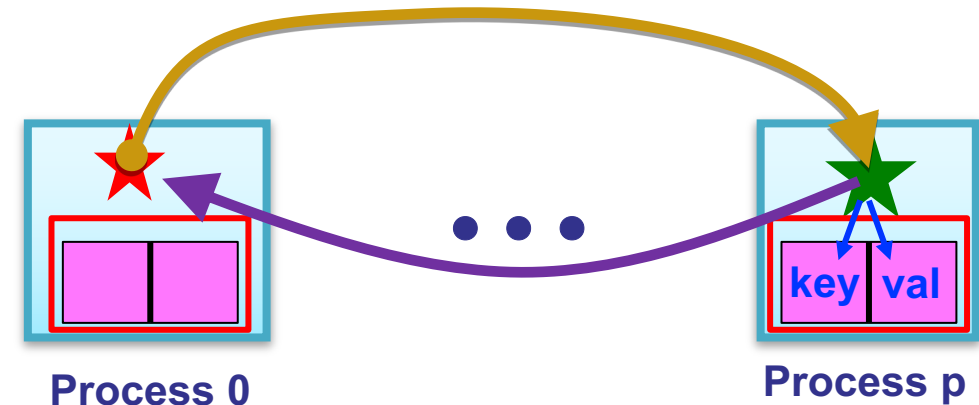
Send RPC to the process determined by key hash

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



Additional DHT operations

// Erases the given key from the DHT.

```
future<> erase(const string &key) {
    return rpc(get_target_rank(key),
               [](dobj_map_t &lmap, const string &key) {
                   lmap->erase(key);
               }, local_map, key);
}
```

Lambda to remove
the key from the local
map at the target

*// Replaces the value associated with the given key and returns the old
// value with which it was previously associated.*

```
future<string> update(const string &key,
                     const string &value) {
    return rpc(get_target_rank(key),
               [](dobj_map_t &lmap, const string &key,
                  const string &value) {
                   return local_update(*lmap, key, value);
               }, local_map, key, value);
}
```

Lambda to
update the key
in the local map
at the target

Helper function to update local map

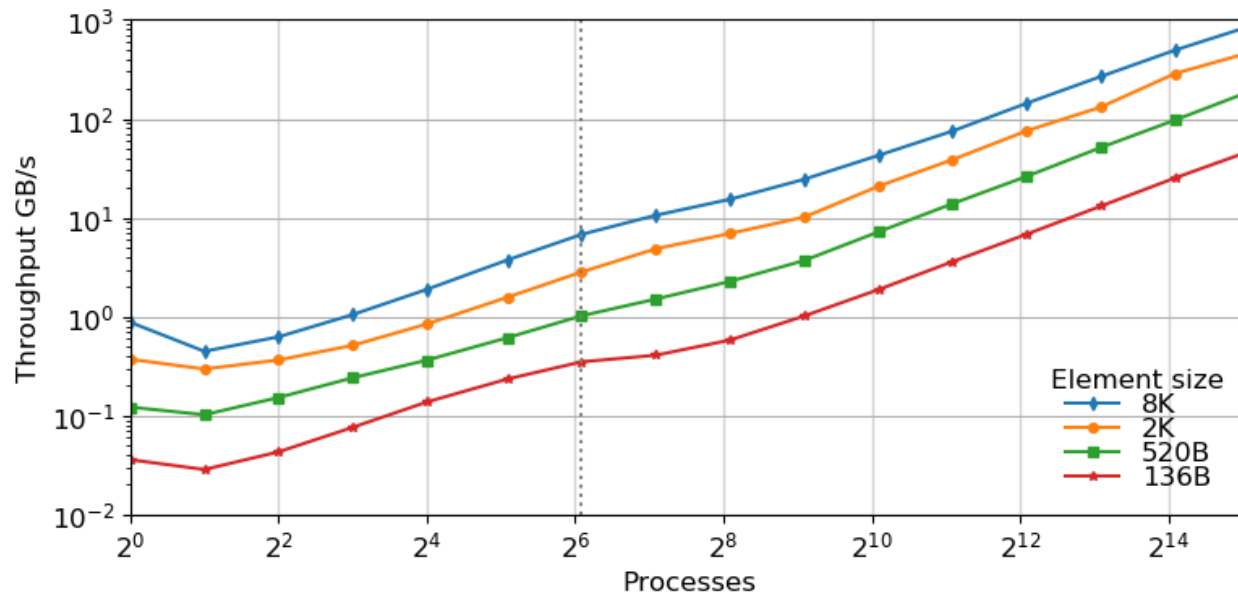
Optimized DHT scales well

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

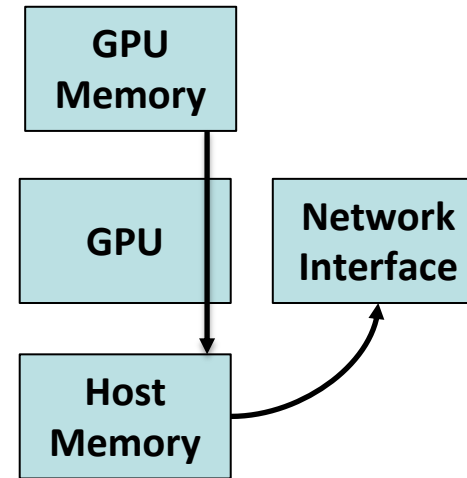
- Team-based barrier, reduction, and broadcast collectives
- Remote atomic operations that utilize hardware offload capabilities of modern networks
- Serialization of complex standard-library and user types in RPC's
- Shared-memory bypass for co-located processes on many-core nodes
- Additional forms 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
- ...

Memory kinds: Accelerated RMA to/from GPU memory

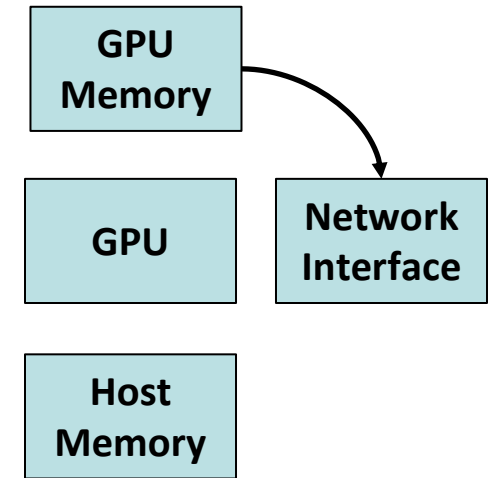
Modern GPUs and NICs can support peer-to-peer data transfers

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)



Data movement
without
acceleration



Data movement
with
acceleration

Memory kinds: Accelerated RMA to/from GPU memory

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

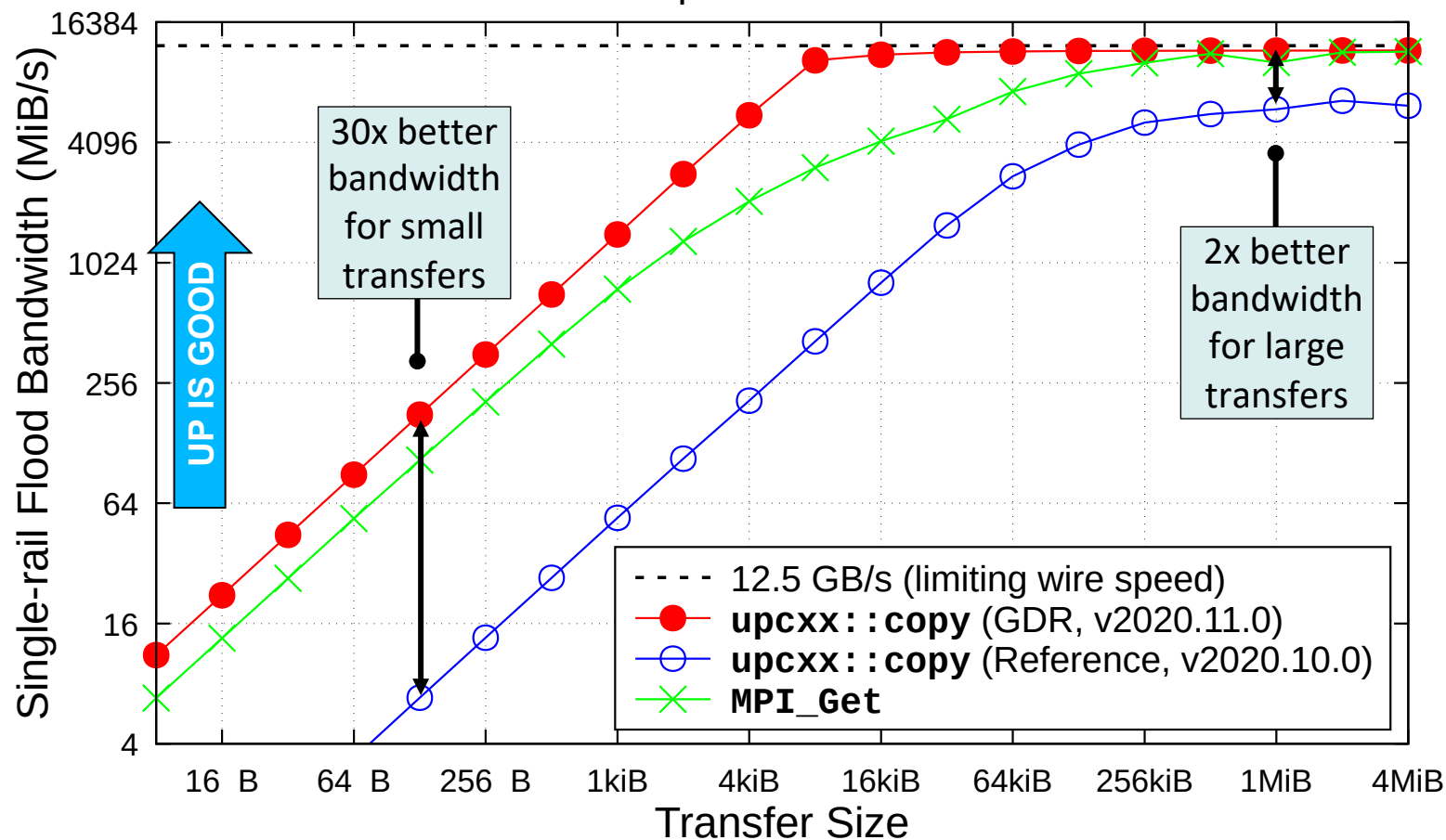
Difference between two consecutive releases shows benefit of GASNet-EX's support for accelerated transfers via Nvidia's "GDR".

- 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

RMA Get Bandwidth (remote GPU to local host memory)
UPC++ 2020.11.0 vs. IBM Spectrum MPI 10.3.1.2 on OLCF Summit

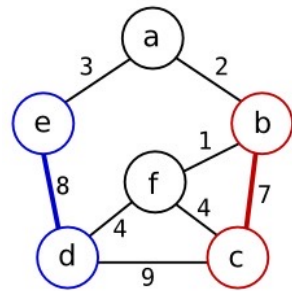
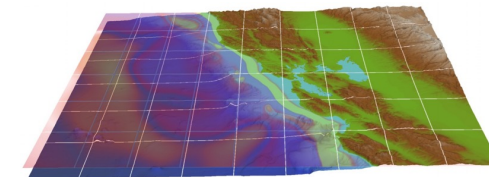
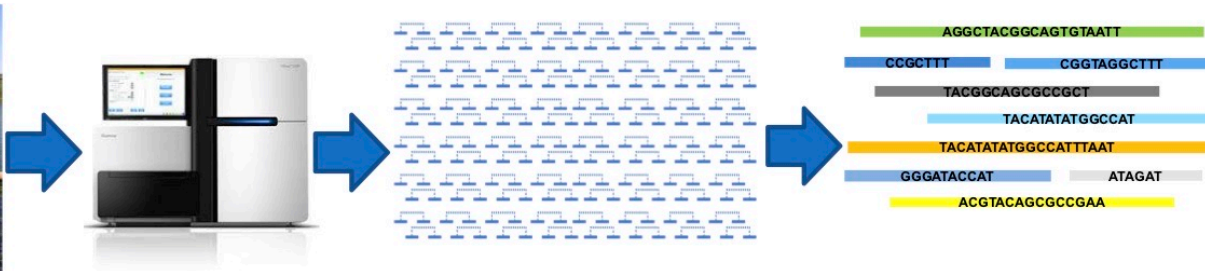
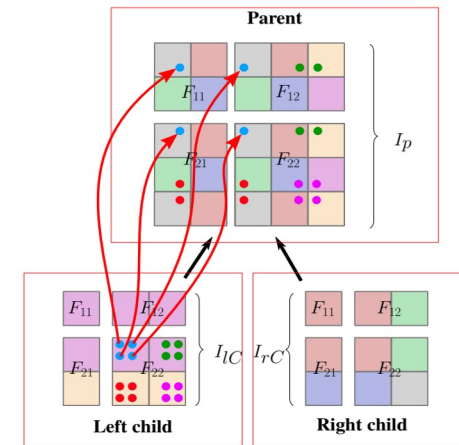
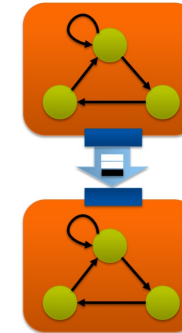
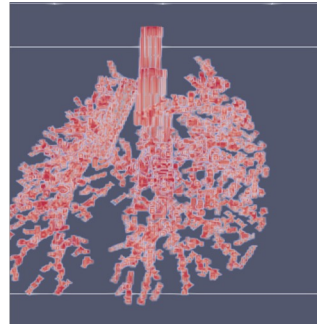


UPC++ results were collected using the version of the `cuda_benchmark` test that appears in the 2020.11.0 release. MPI results are from `osu_get_bw` test in a CUDA-enabled build of OSU Micro-Benchmarks 5.6.3. All tests were run on OLCF Summit, between two nodes with one process per node, over its EDR InfiniBand network.

UPC++ applications

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

- symPack, a sparse symmetric matrix solver
- 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



symPACK: UPC++ provides productivity + performance

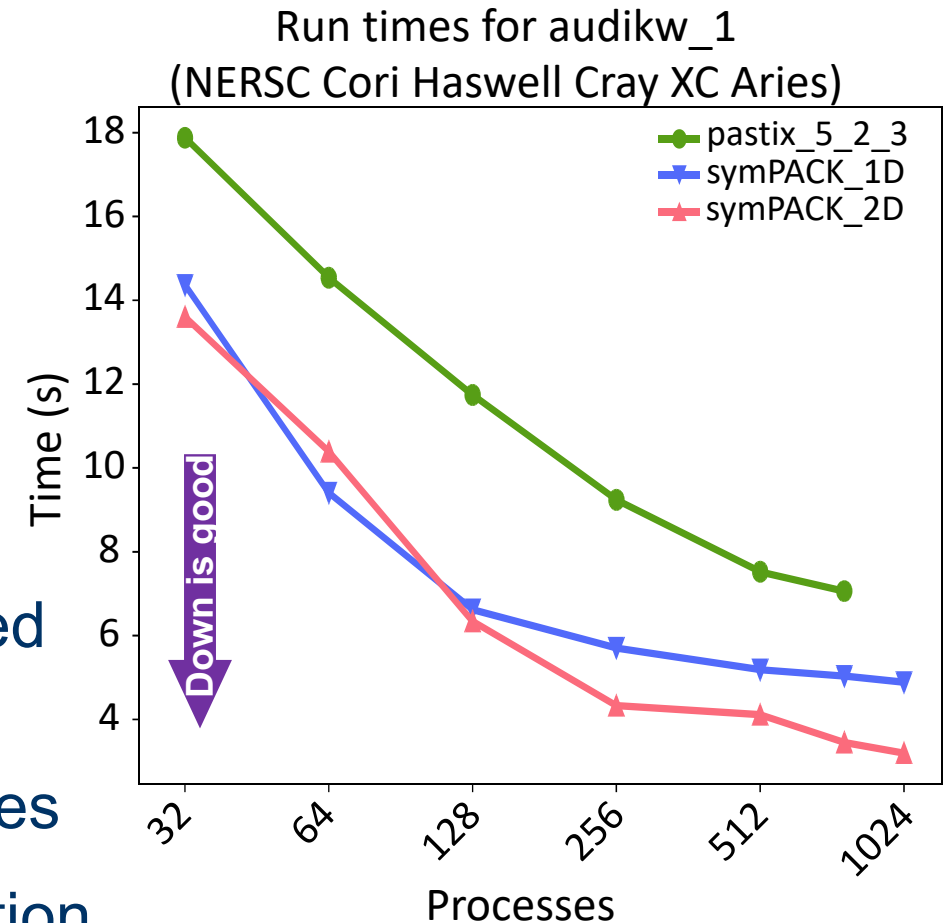
Productivity

- 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>



SIMCoV: Spatial Model of Immune Response to Viral Lung Infection

Model the entire lung at the cellular level:

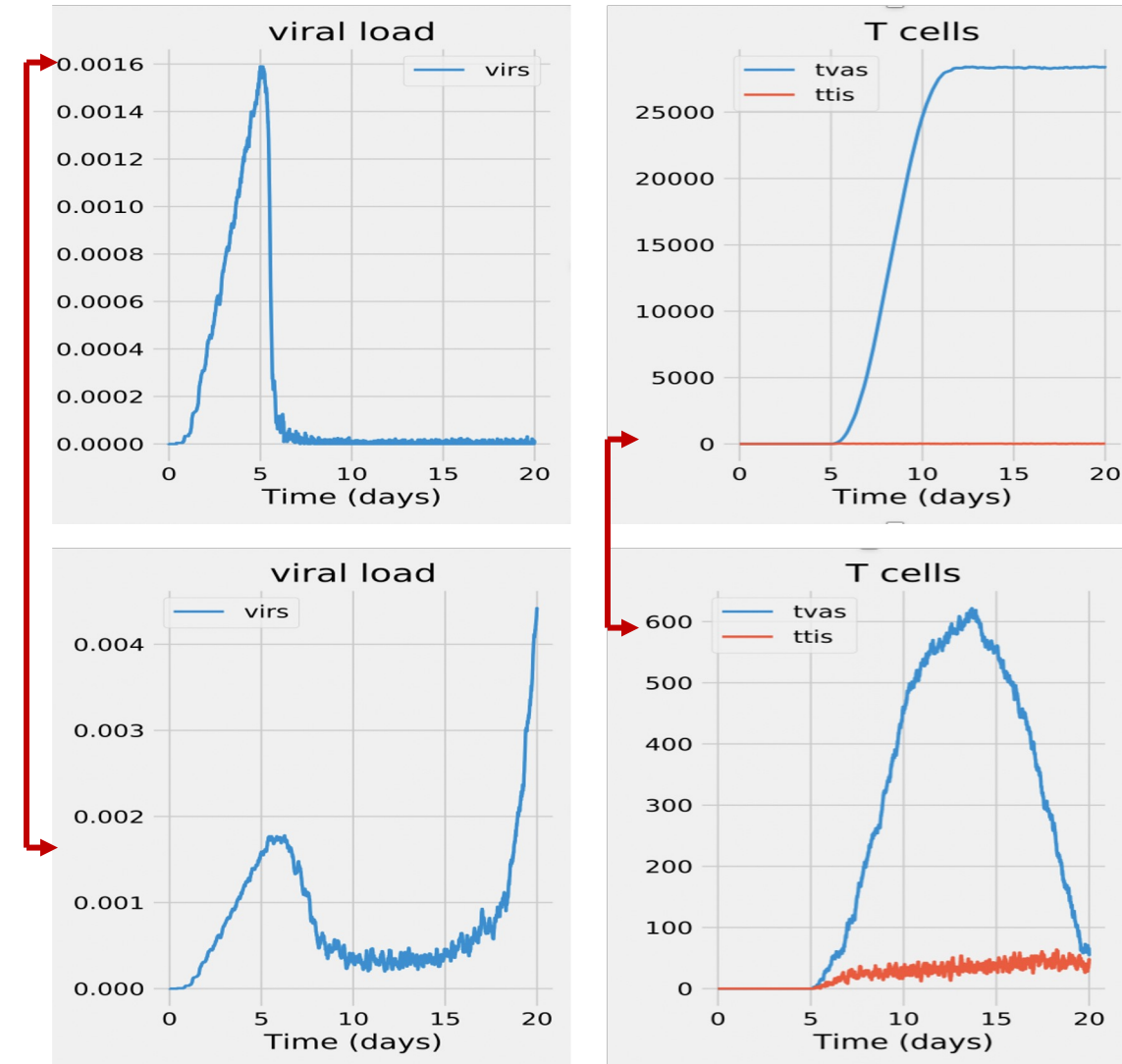
- 100 billion epithelial cells
- 100s of millions of T cells
- Complex branching fractal structure
- Time resolution in seconds for 20 to 30 days

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

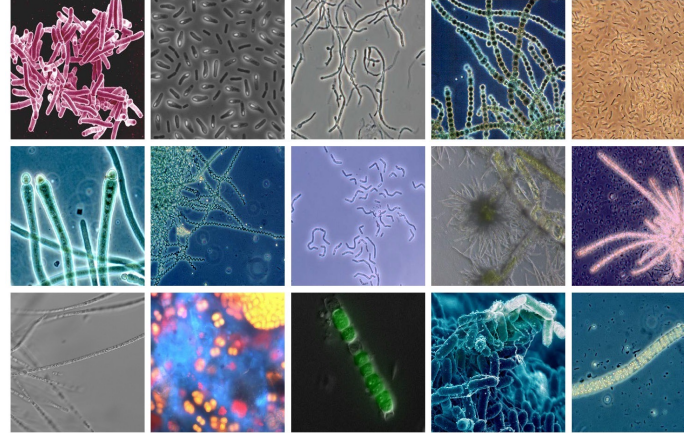


<https://github.com/AdaptiveComputationLab/simcov>

ExaBiome: Exascale Solutions for Microbiome Analysis



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



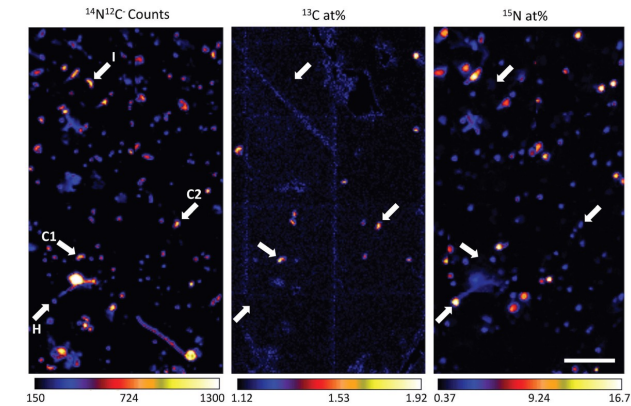
What are the microbial dynamics of soil carbon cycling? (3.3 TB)



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



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

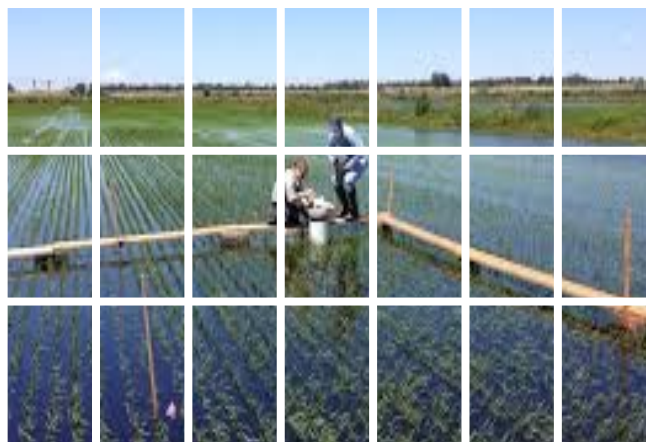


Combine genomics with isotope tracing methods for improved functional understanding (8TB)

Co-Assembly improves quality and is an HPC problem

Full wetlands data: 2.6 TB of data in 21 lanes (samples)

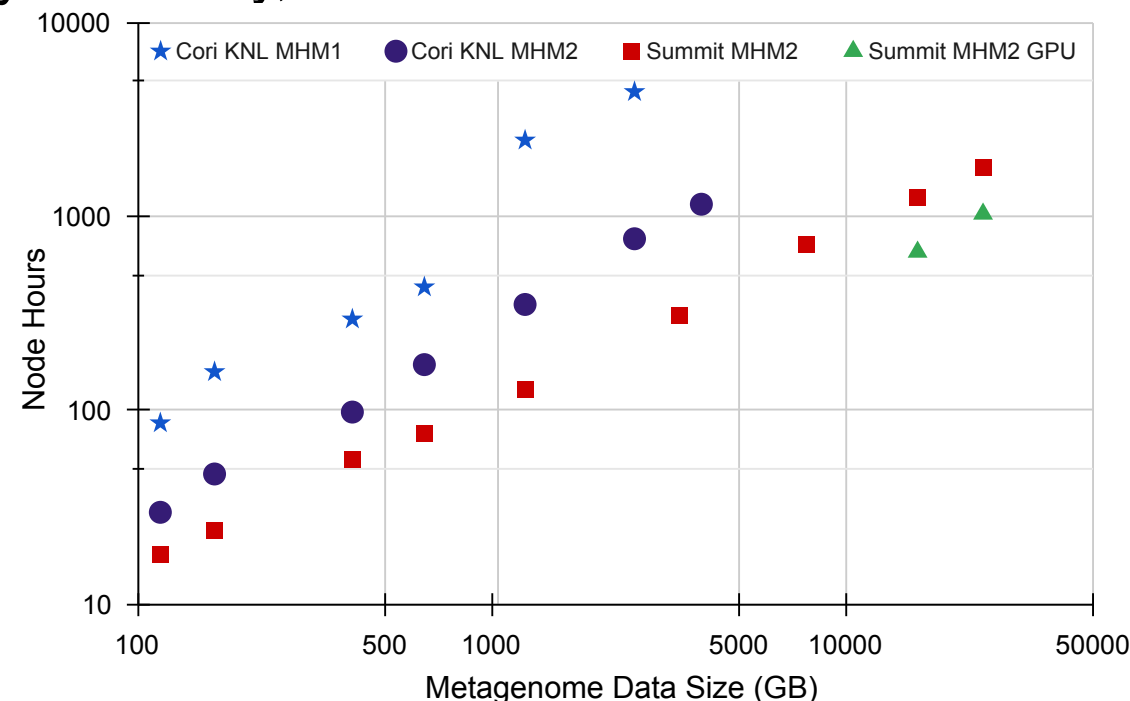
- Time-series samples from multiple sites of Twitchell Wetlands in the San Francisco Bay-Delta
- Previously assembled 1 lane at a time (multiassembly)
- MetaHipMer coassembled together – higher quality assembly, in **3.5 hours on 16K cores**



Multiassembly
1 lane at a time



Coassembly all assembled
together – more new genomes
at higher completeness



This was the largest, high-quality de novo metagenome assembly completed at the time

More recently: new record 30TB metagenome assembly on 1500 nodes (63K cores and 9K GPUs) of OLCF Summit in 2022

Evangelos Georganas, Rob Egan, Steven Hofmeyr, Eugene Goltsman, Bill Arndt, Andrew Tritt, Aydin Buluc, Leonid Oliker, Katherine Yelick, **SC18 best paper finalist**

MetaHipMer utilized UPC++ features

C++ templates – efficient code reuse

dist_object – as a templated functor & data store

Asynchronous all-to-all exchange – not batch synchronous

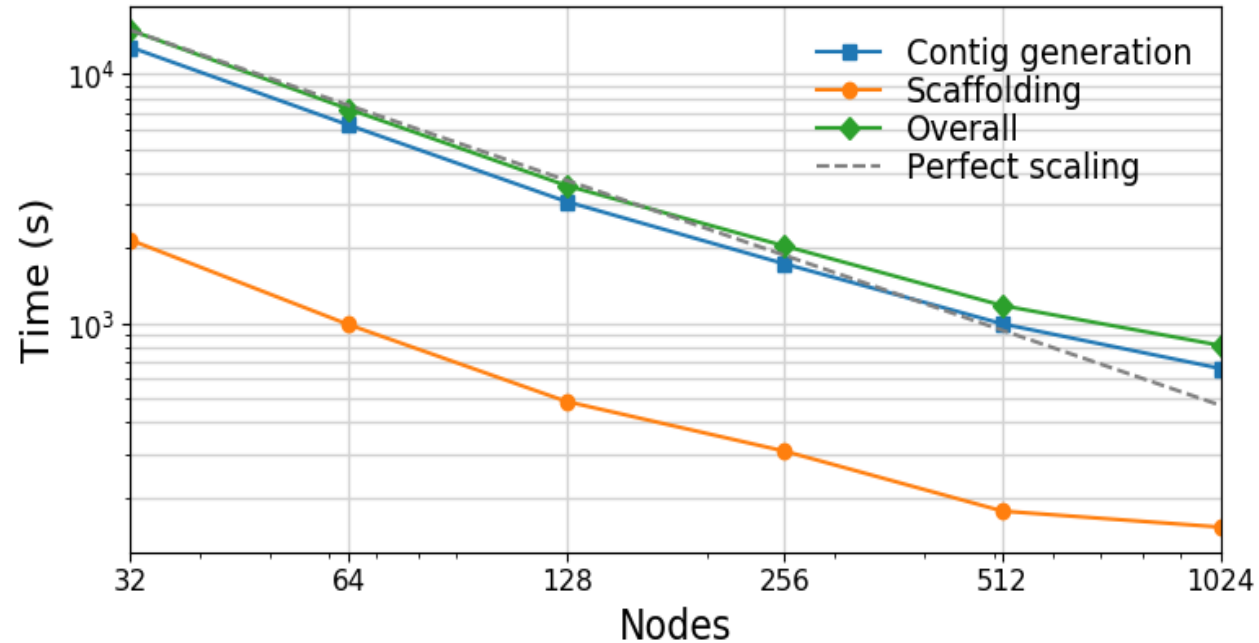
- 5x improvement at scale relative to previous MPI implementation

Future-chained workflow

- Multi-level RPC messages
- Send by node, then by process

Promise & fulfill (advanced UPC++ feature) – for a fixed-size memory footprint

- Issue promise when full, fulfill when available



*Work and results by Rob Egan,
funded by ECP ExaBiome Group*

<https://sites.google.com/lbl.gov/exabiome/downloads>

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 Fraguera 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