BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

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

Amir Kamil
upcxx.lbl.gov
pagoda@lbl.gov
https://upcxx.lbl.gov/training

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, Rob Egan, Max Grossman, Paul H. Hargrove, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Erich Strohmaier, Daniel Waters, Katherine Yelick

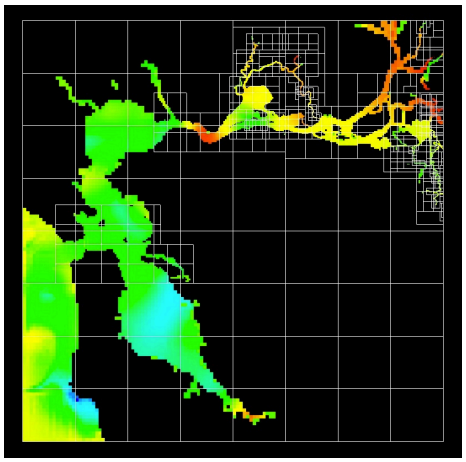
This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including 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.

Some motivating applications

PGAS well-suited to applications that use irregular data structures

- Sparse matrix methods
- Adaptive mesh refinement
- Graph problems, distributed hash tables

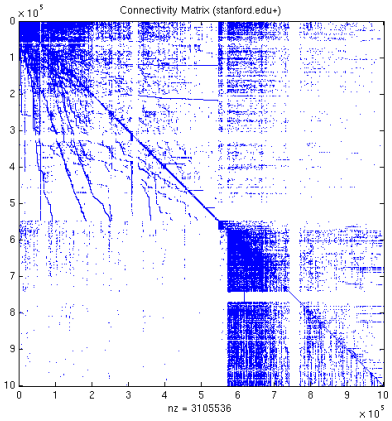


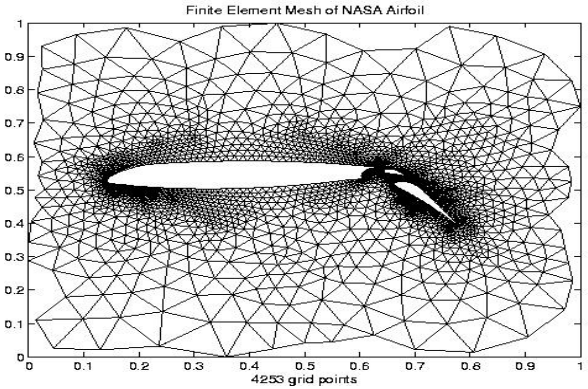
http://tinyurl.com/yxqarenl

Processes may send different amounts of information to other processes

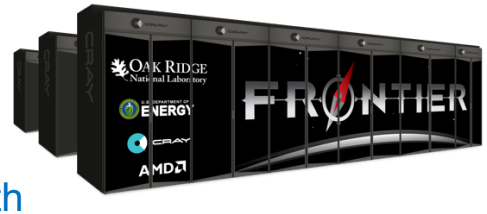
The amount can be data dependent, dynamic

Courtesy of Jim Demmel





The impact of fine-grained communication



The first exascale systems will appear soon

- In the USA: **Frontier** (2021) https://tinyurl.com/y2ptx3th

Some apps employ *fine-grained* communication

- Messages are short, so the overhead term dominates communication time $\alpha + F(\beta^{-1}_{\infty}, n)$
- They are latency-limited, and latency is only improving slowly

Memory per core is dropping, an effect that can force more frequent fine-grained communication

We need to reduce communication costs

- **Asynchronous communication and execution are critical**
- But we also need to keep overhead costs to a minimum

Reducing communication overhead

What if we could let each process directly access one another's memory via a global pointer?

- We don't need to match sends to receives
- We don't need to guarantee message ordering
- There are no unexpected messages

Communication is **one-sided**

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

Looks like shared memory

Observation: modern network hardware provides the capability to directly access memory on another node: Remote Direct Memory Access (RDMA)

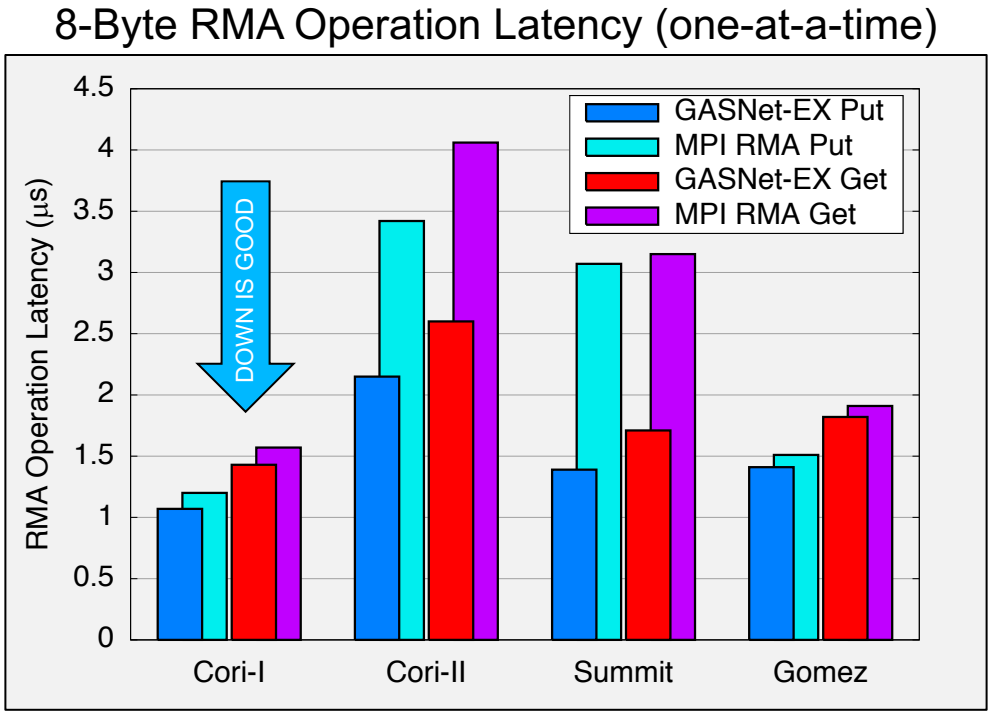
- Can be compiled to load/store if source and target share physical memory

RMA performance: GASNet-EX vs MPI-3

Three different MPI implementations

Two distinct network hardware types

On these four systems the performance of GASNet-EX meets or exceeds MPI RMA:

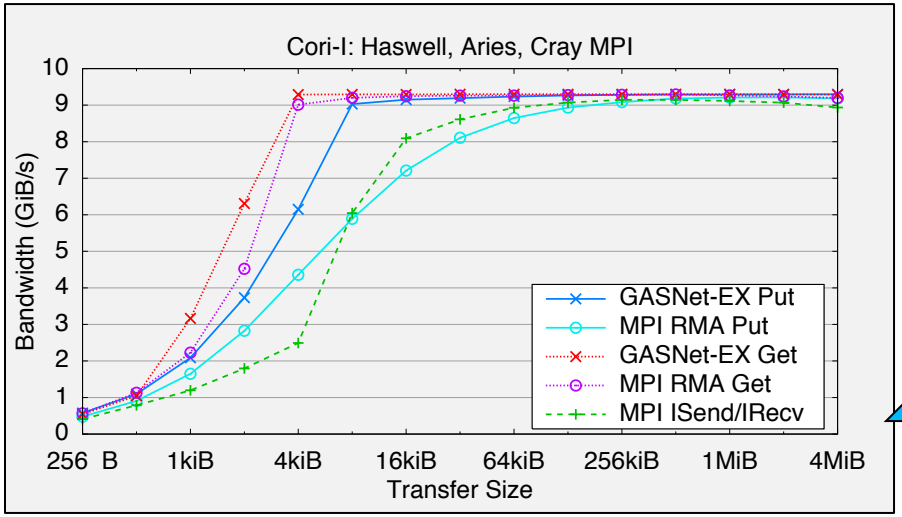


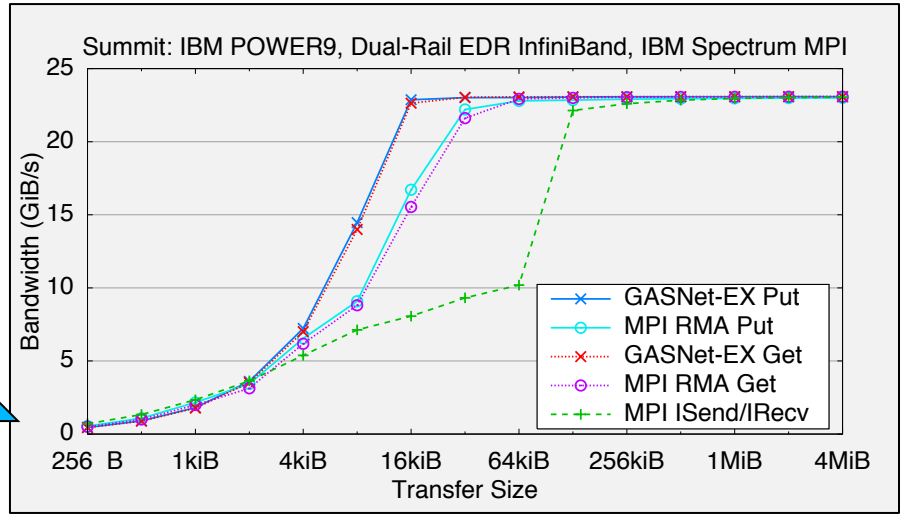
- 8-byte Put latency 6% to 55% better
- 8-byte Get latency 5% to 45% better
- Better flood bandwidth efficiency, typically saturating at ½ or ¼ the transfer size (next slide)

GASNet-EX results from v2018.9.0 and v2019.6.0. MPI results from Intel MPI Benchmarks v2018.1.
For more details see Languages and Compilers for Parallel Computing (LCPC'18). **https://doi.org/10.25344/S4QP4W**
More recent results on Summit here replace the paper's results from the older Summitdev.

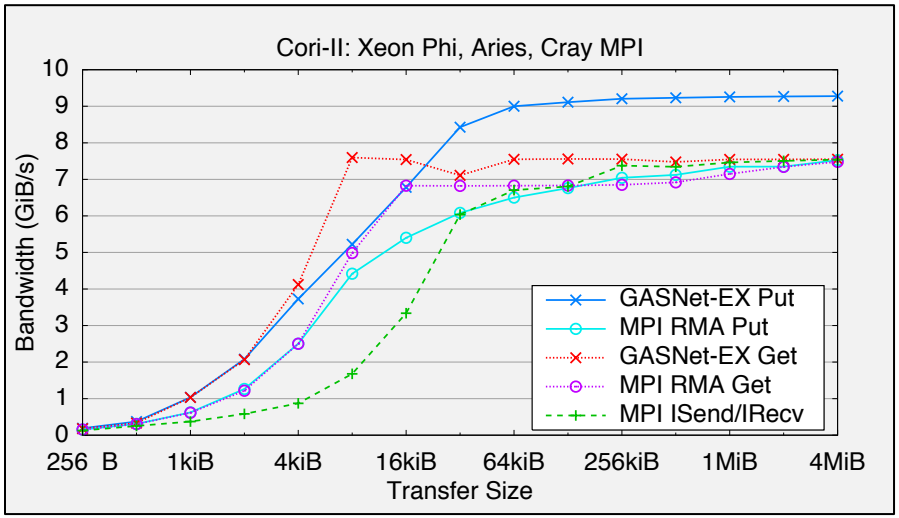
RMA performance: GASNet-EX vs MPI-3

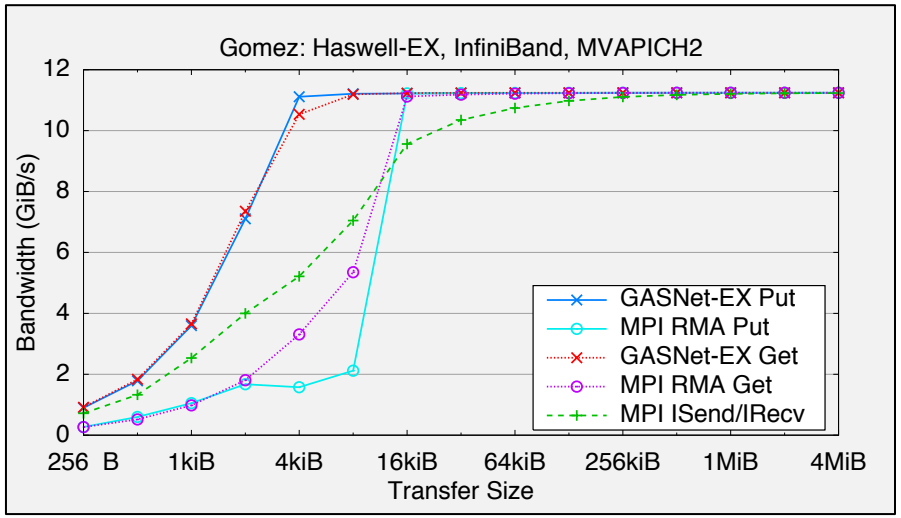
Uni-directional Flood Bandwidth (many-at-a-time)





UP IS GOOD

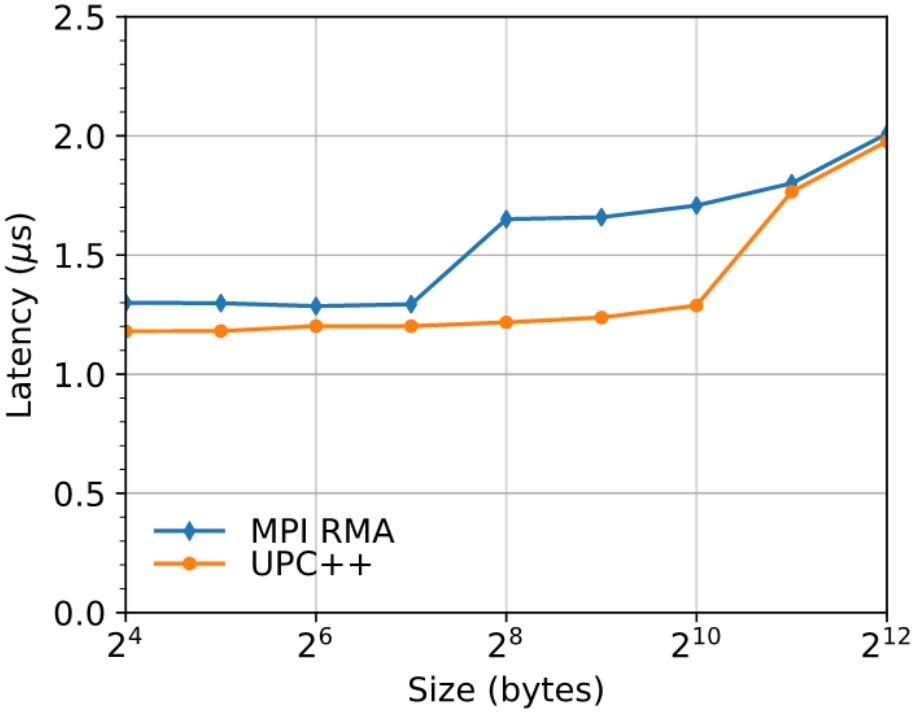


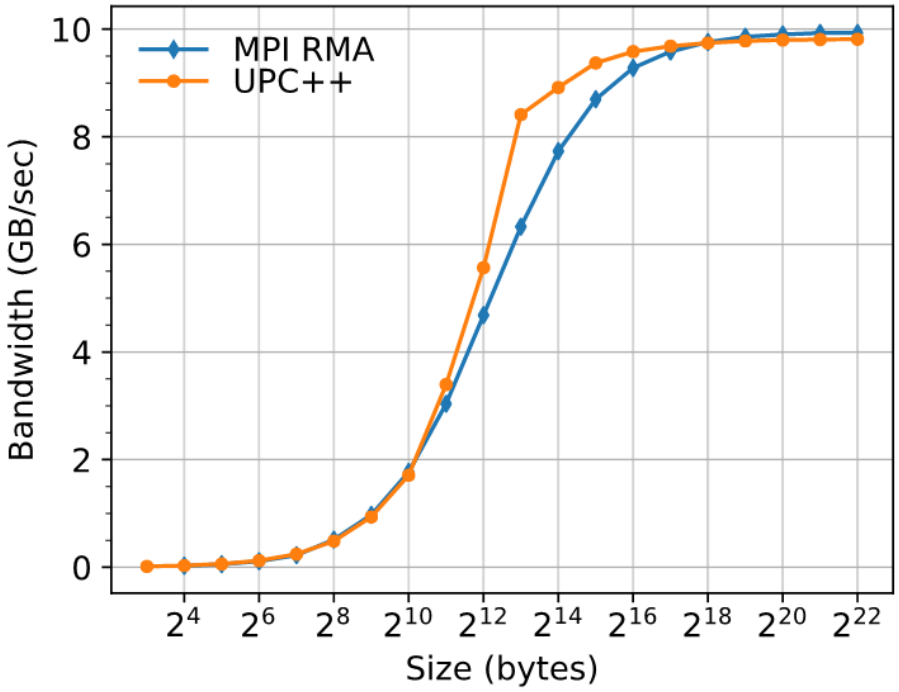


RMA microbenchmarks

Experiments on NERSC Cori:
- Cray XC40 system

Two processor partitions:
- Intel Haswell (2 x 16 cores per node)
- Intel KNL (1 x 68 cores per node)





Round-trip Put Latency (lower is better)

Flood Put Bandwidth (higher is better)

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

The PGAS model

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

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

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

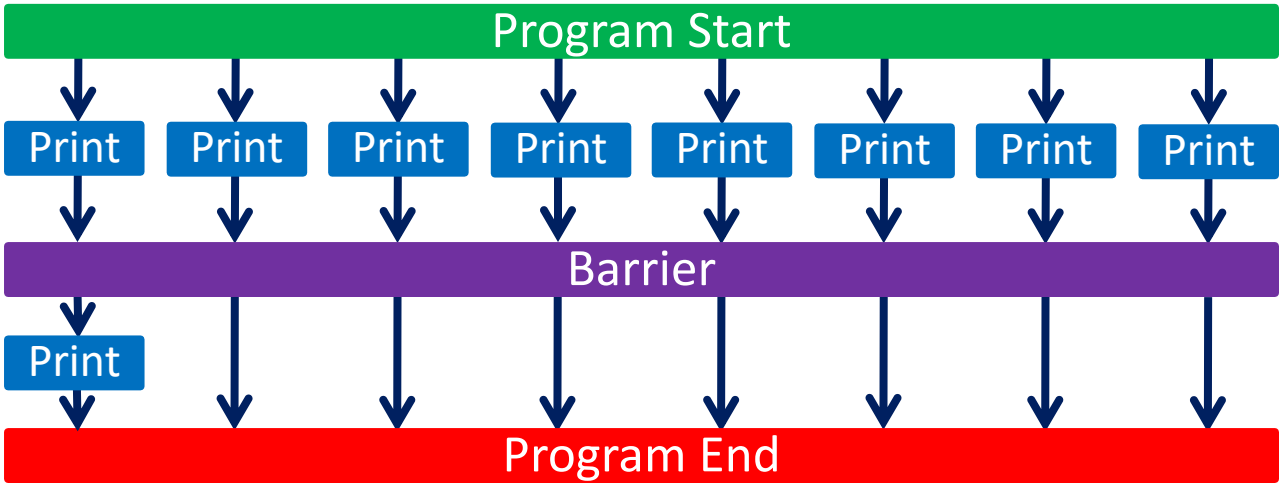
Libraries that support PGAS: Habanero UPC++, OpenSHMEM, Co-Array C++, Global Arrays, DASH, MPI-RMA

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

Execution model: SPMD

Like MPI, 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();
}
```



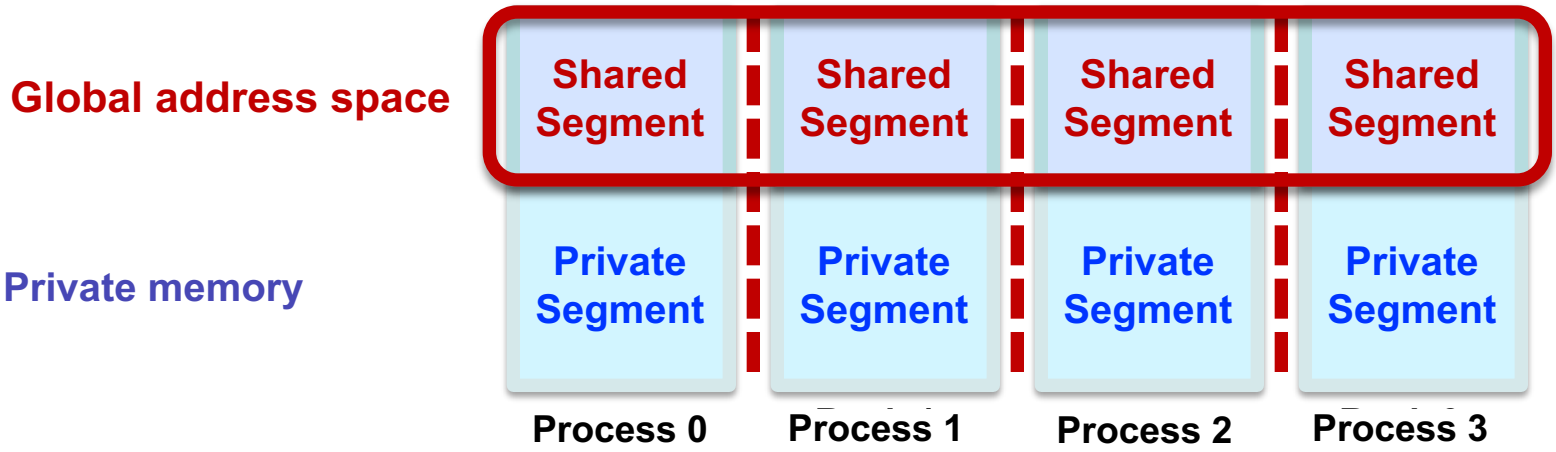
A Partitioned Global Address Space

Global Address Space

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

Partitioned

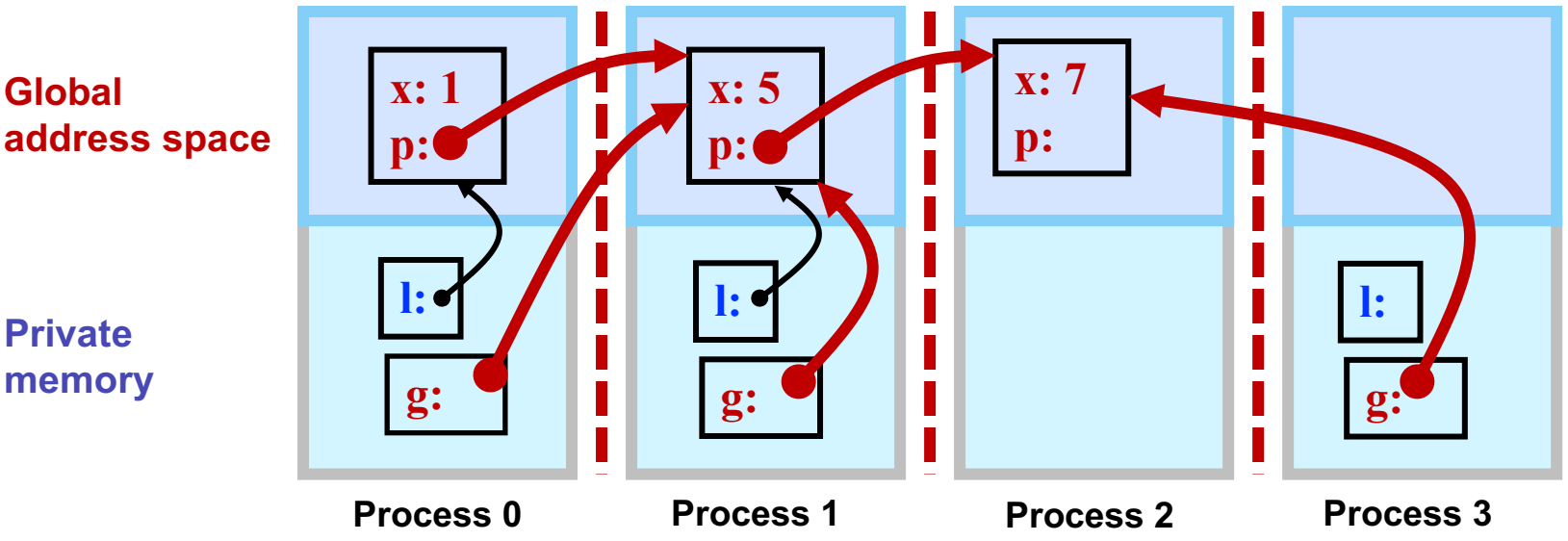
- *Global pointers* to objects in shared memory have an affinity to a particular process
- Explicitly managed by the programmer to optimize for locality
- In conventional shared memory, pointers do not encode affinity



Global vs. raw pointers

We can create data structures with embedded global pointers

Raw C++ pointers can be used on a process to refer to objects in the global address space that have affinity to that process

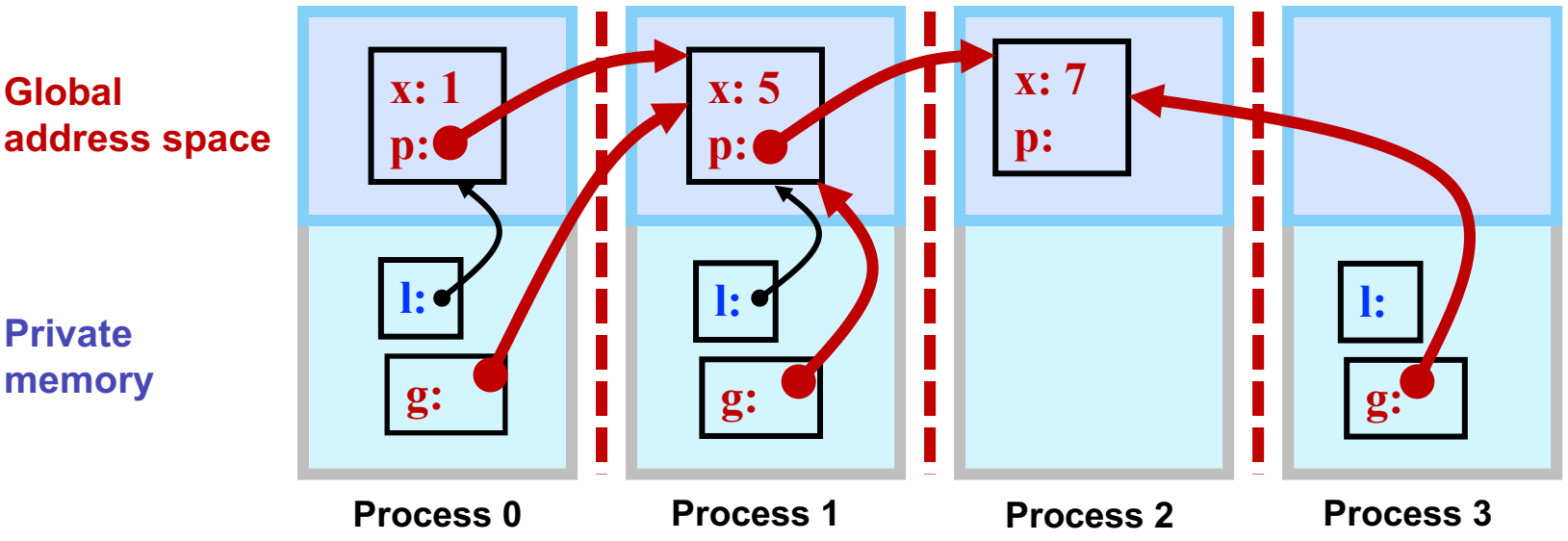


What is a global pointer?

A global pointer carries both an address and the affinity for the data

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

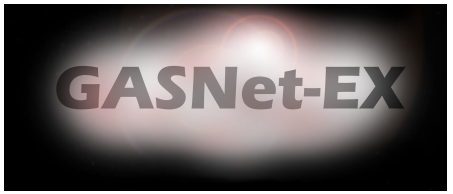
The affinity identifies the process that created the object



How does UPC++ deliver the PGAS model?

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 the network, whatever that network may be, including any special-purpose offload support
- Active messages efficiently support Remote Procedure Calls (RPCs), which are expensive to implement in other models
- Enables portability (laptops to supercomputers)

Designed to allow interoperation with existing programming systems

- Same process model as MPI, enabling hybrid applications
- OpenMP and CUDA can be mixed with UPC++ in the same way as MPI+X

upC++ BERKELEY LAB

What does UPC++ offer?

Asynchronous behavior based on futures/promises

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

Design principles encourage performant program design

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

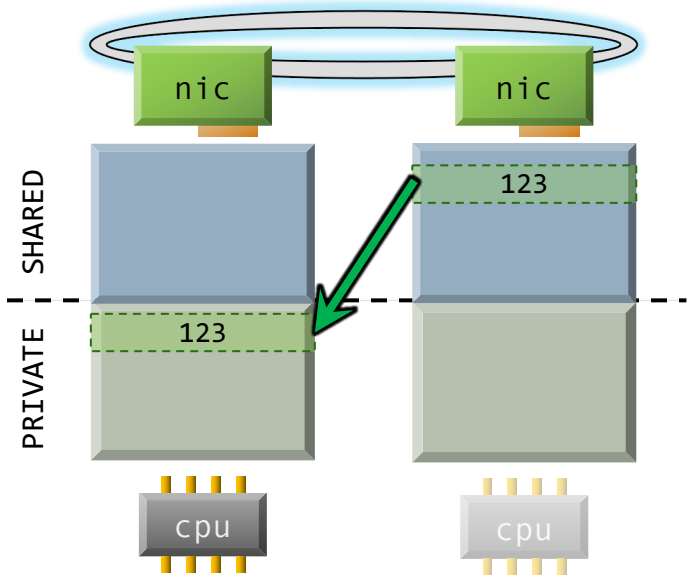
Asynchronous communication

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

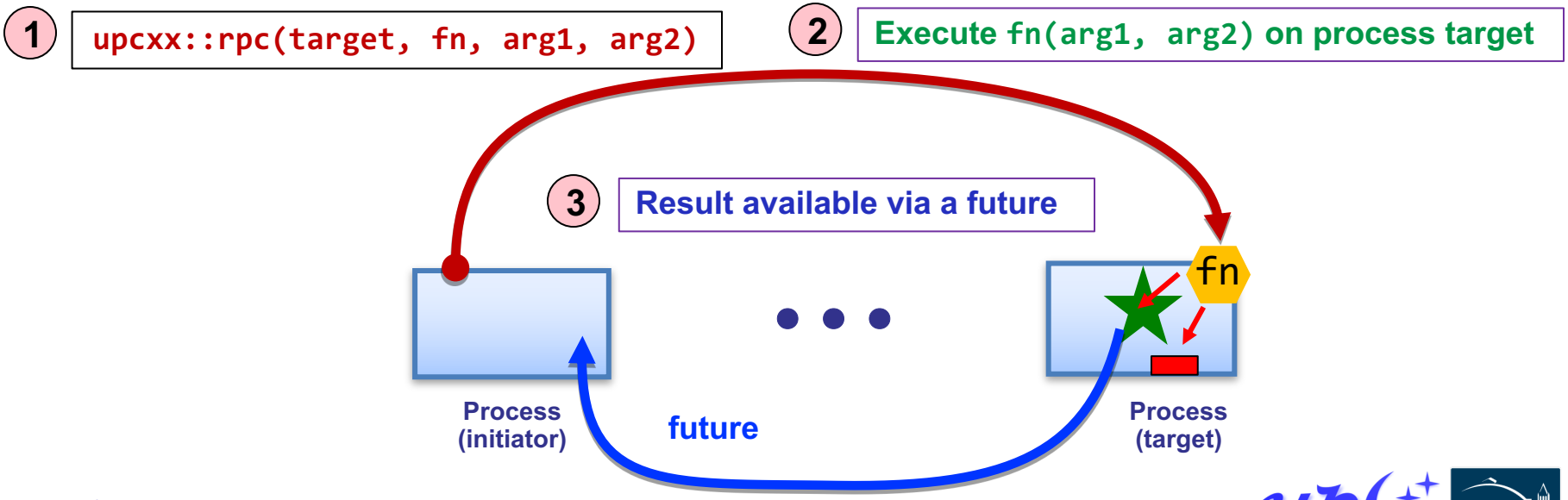


Remote procedure call

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



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
```

Using UPC++ at DOE Centers

ALCF's Theta

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

NERSC's Cori

```
$ module load upcxx
```

OLCF's Summit

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

More info and examples for all three centers are available from **upcxx.lbl.gov/wiki/docs/site-docs**

Works on laptops, workstations and clusters too.

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 (such as `-I`, `-L`, `-l`).
- We also provide other mechanisms for compiling (`upcxx-meta`, CMake package).

Launch wrapper:

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

- Arguments similar to other familiar tools
- We also support launch using platform-specific tools, such as `srun`, `jsrun` and `aprun`.

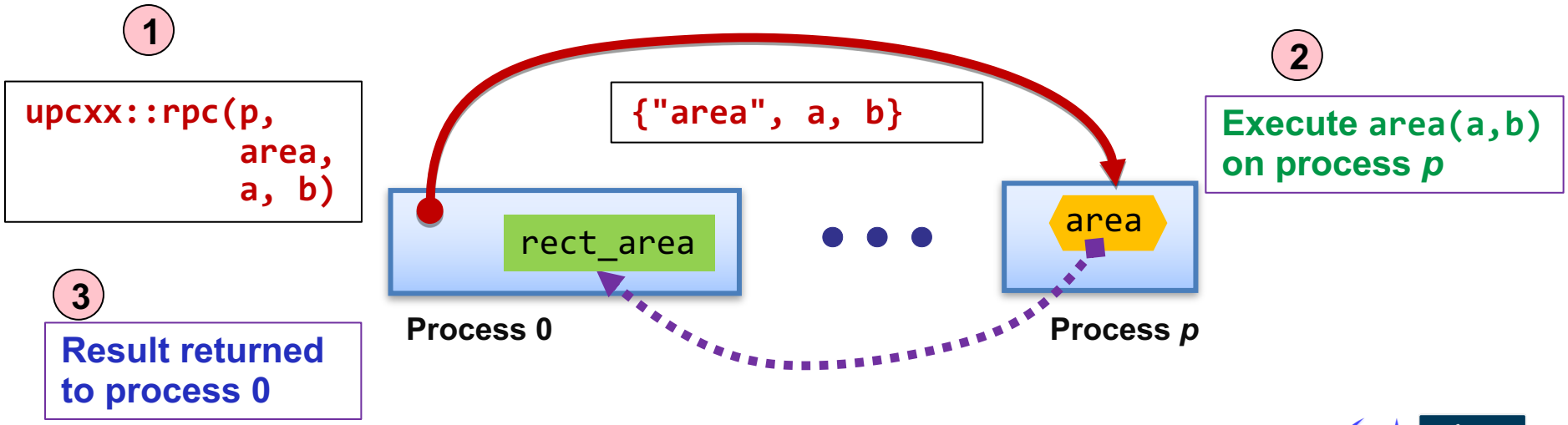
Remote Procedure Calls (RPC)

Let's say that process 0 performs this RPC

```
int area(int a, int b) { return a * b; }

int rect_area = rpc(p, area, a, b).wait();
```

The target process *p* will execute the handler function `area()` at some later time determined at the target

The result will be returned to process 0



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();
```

What is a future?

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

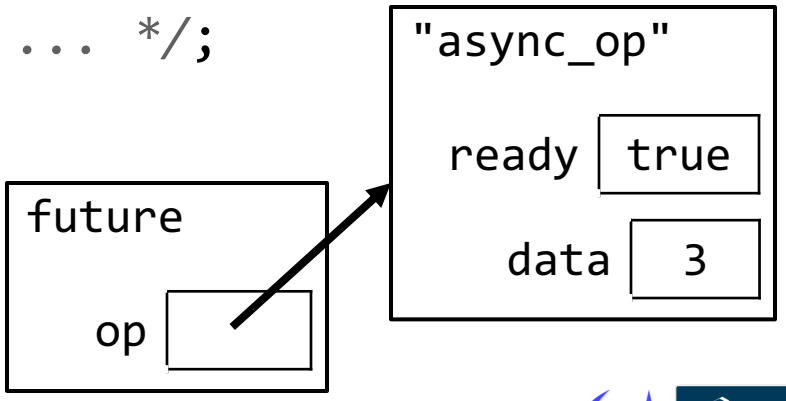
- The status of the operation
- The results (zero or more values) of the completed operation

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++

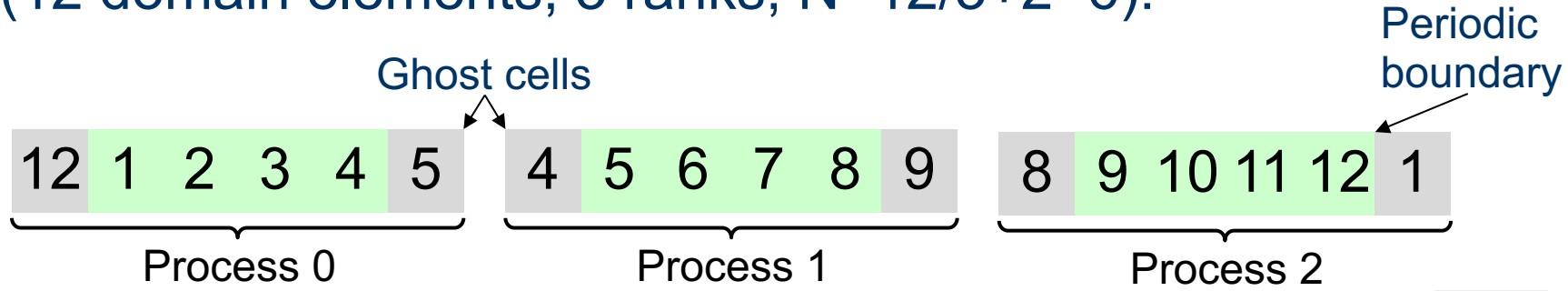
Iterative algorithm that updates each grid cell as a function of its old value and those of its immediate neighbors

Out-of-place computation requires two grids

Local grid size

```
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 ranks, N=12/3+2=6):

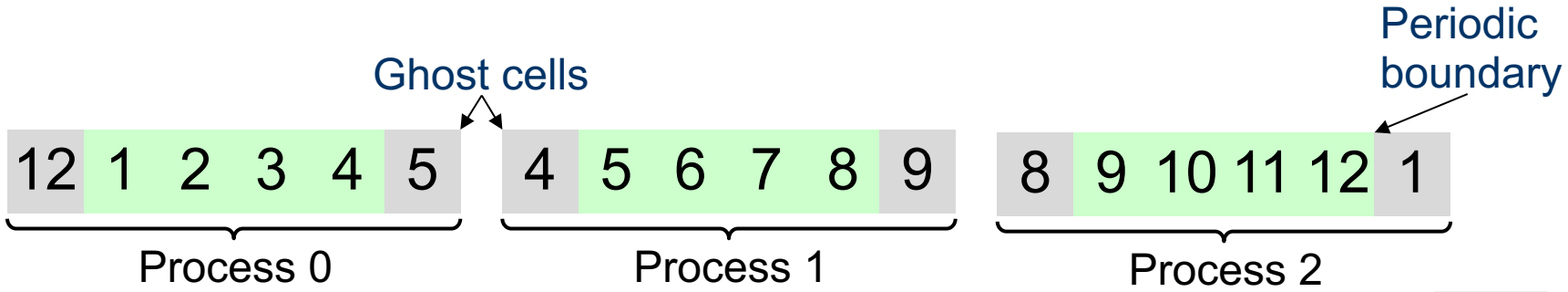


Jacobi boundary exchange (version 1)

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)

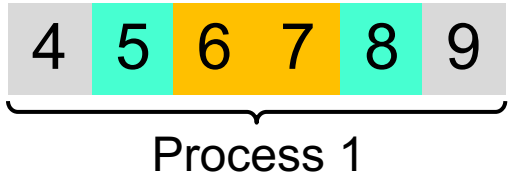
We can use RPC to communicate boundary cells

```
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);
```

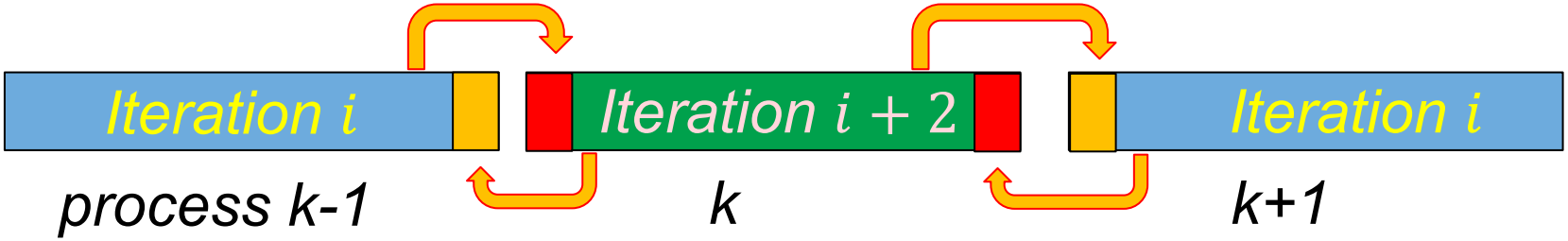


Race conditions

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:

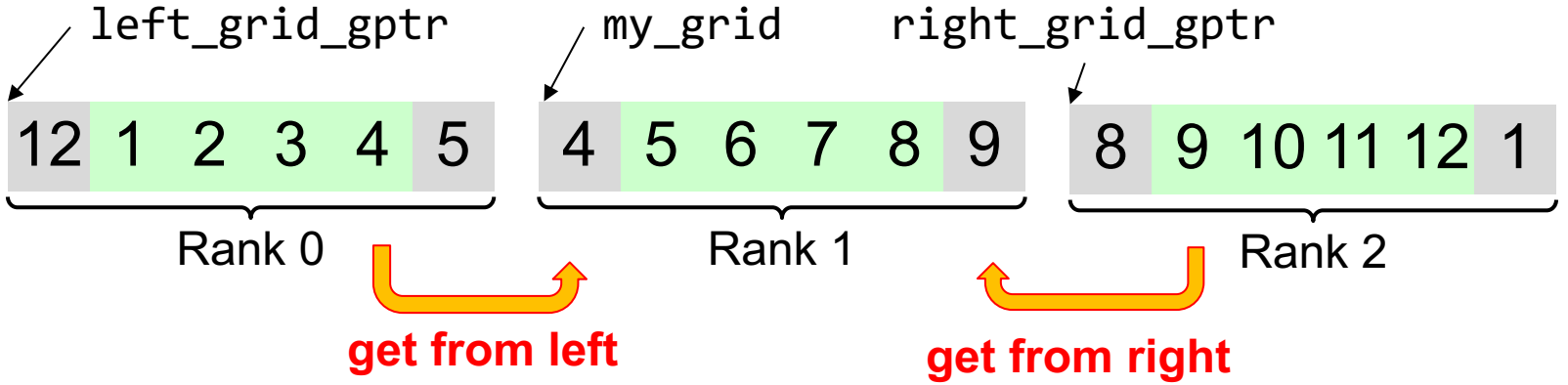
```
global_ptr<int> remote = /* ... */;
future<int> fut1 = rget(remote);
int result = fut1.wait();
future<> fut2 = rput(42, remote);
fut2.wait();
```

- 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

UPC++ does not maintain a symmetric heap

- 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;
```

Can be accessed by an RPC

```
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();
}
```

Later, we will see how downcasting can be used with processes that share physical memory

Jacobi RMA with gets

Each process obtains boundary data from its neighbors with rget()

```
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();
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]);
```

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);
  });
```

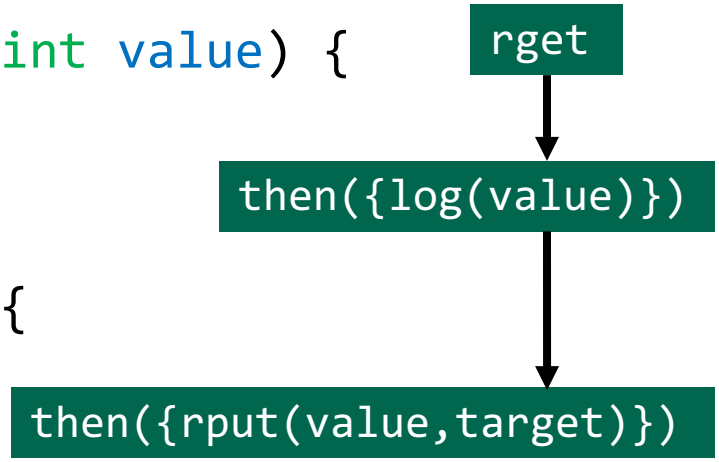
Vector get does not produce a value

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();
```



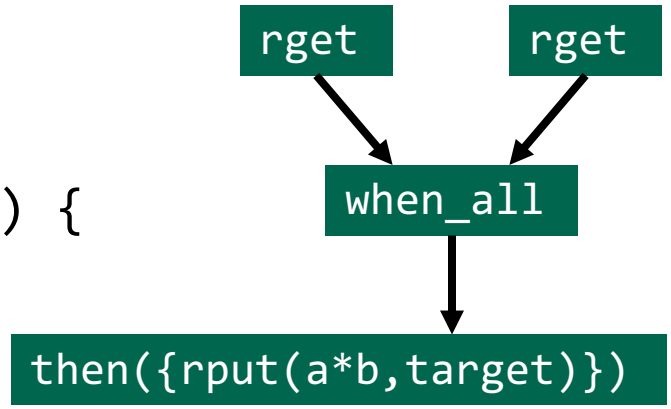
This code retrieves an integer from a remote location, computes its log, and then sends it to a different remote location

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();

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]);
```

Ensure outgoing puts have completed

Ensure incoming puts have completed

Distributed objects

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

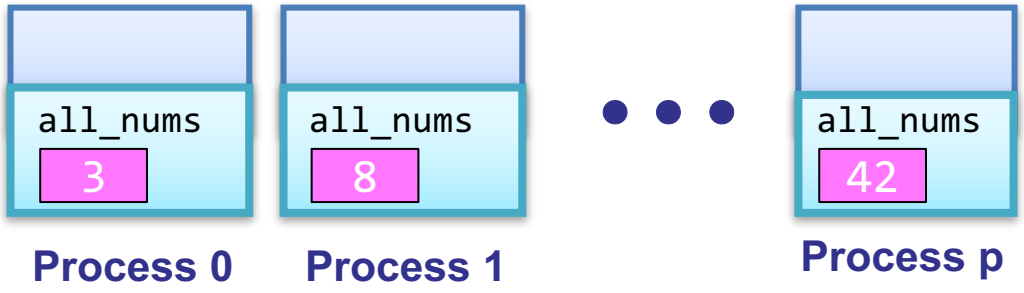
```
dist_object<T>(T value, team &team = world());
```

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
- Can be constructed over teams

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



Bootstrapping the communication

Since allocation is not collective, we must arrange for each process to obtain pointers to its neighbors' grids

We can use a distributed object to do so

```
using ptr_pair = std::pair<global_ptr<double>,
                           global_ptr<double>>;
dist_object<ptr_pair> dobj({old_grid_gptr,
                            new_grid_gptr});
std::tie(right_old_grid, right_new_grid) =
  dobj.fetch(right).wait();
// equivalent to the statement above:
//   ptr_pair result = dobj.fetch(right).wait();
//   right_old_grid = result.first;
//   right_new_grid = result.second;

barrier();
```

Ensures distributed objects are not destructed until all ranks have completed their fetches

Implicit synchronization

The future returned by `fetch()` is not readied until the distributed object has been constructed on the target, allowing its value to be read

- This allows us to avoid explicit synchronization between the creation and the `fetch()`

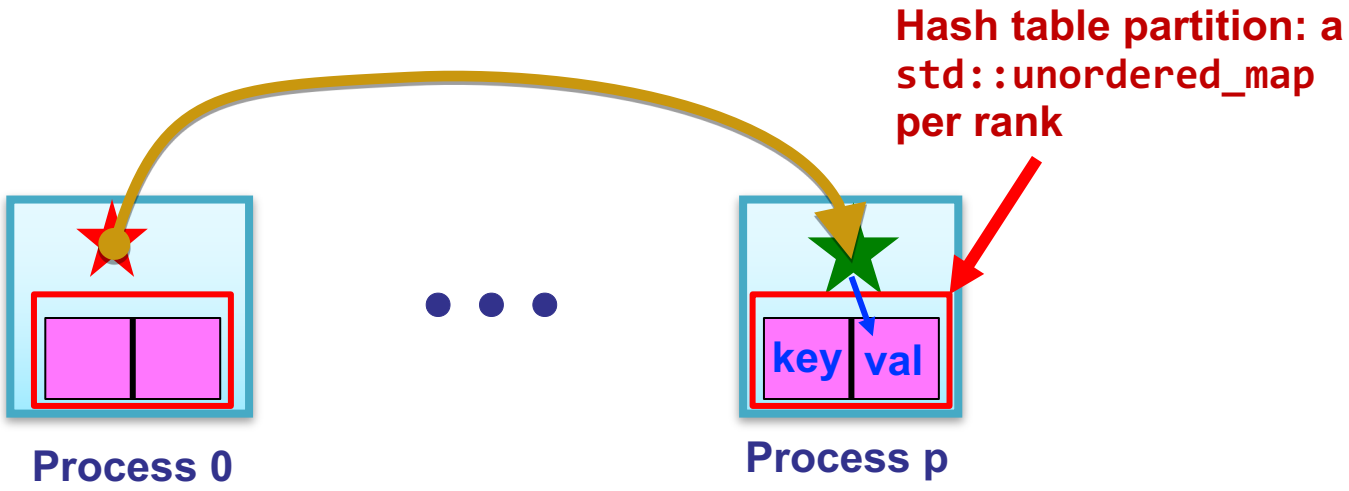
```
using ptr_pair = std::pair<global_ptr<double>,
                           global_ptr<double>>;
dist_object<ptr_pair> dobj({old_grid_gptr,
                            new_grid_gptr});

std::tie(right_old_grid, right_new_grid) =
  dobj.fetch(right).wait();

barrier();
```

The result of `fetch()` is obtained after the `dist_object` is constructed on the target

Distributed hash table (DHT)

Distributed analog of `std::unordered_map`

- Supports insertion and lookup
- We will assume the key and value types are `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<unordered_map<string, string>>;

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

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

Computes owner for the given key

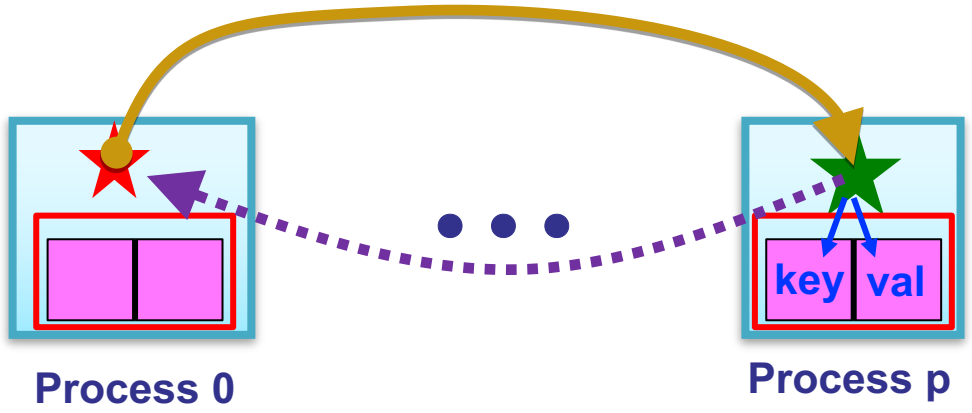
DHT insertion

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),
    [](dobj_map_t &lmap, const string &key,
       const string &val) {
      (*lmap)[key] = val;
    }, local_map, key, val);
}
```

Key and value passed as arguments to the remote function

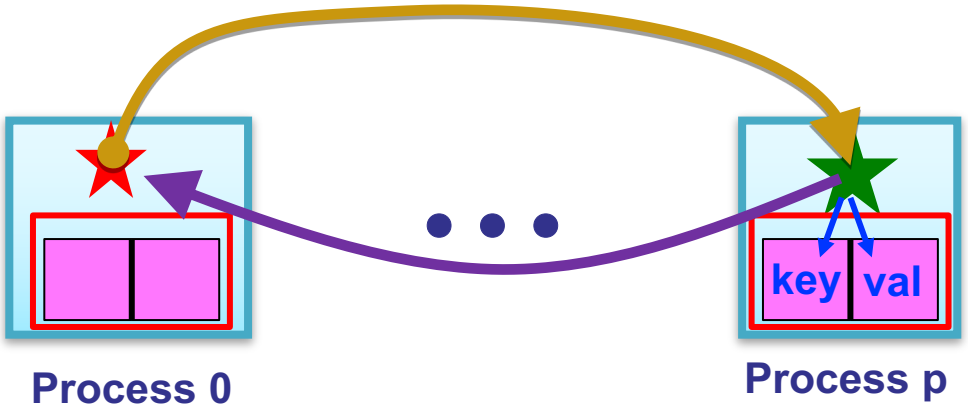
UPC++ uses the distributed object's universal name to look it up on the remote process



DHT find

Find also uses RPC and returns a future

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



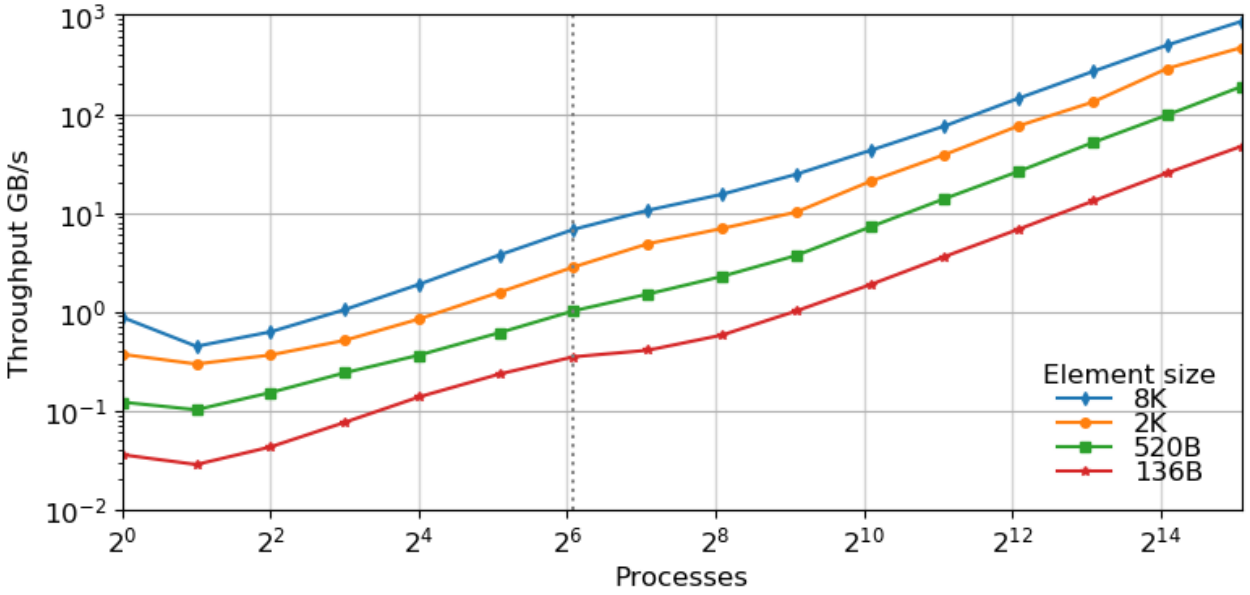
Optimized DHT scales well

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)

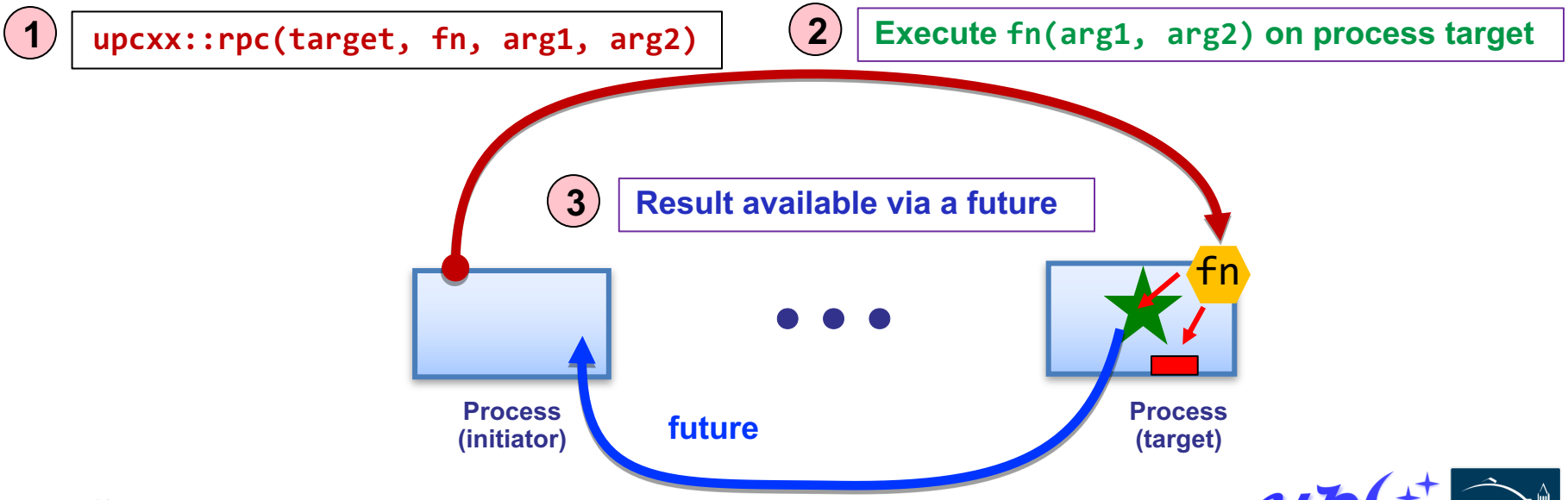
Cray XC40

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 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 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});
```

- Support `int64_t`, `int32_t`, `uint64_t`, `uint32_t`, `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 *serialize* shipped data

- Conversion between in-memory and byte-stream representations
- serialize (sender) → transfer → deserialize → invoke (target)

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>>>, ...`
- 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

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

Process elements directly from the network buffer

Cheap view construction

Shared memory hierarchy and `local_team`

Memory systems on supercomputers are hierarchical

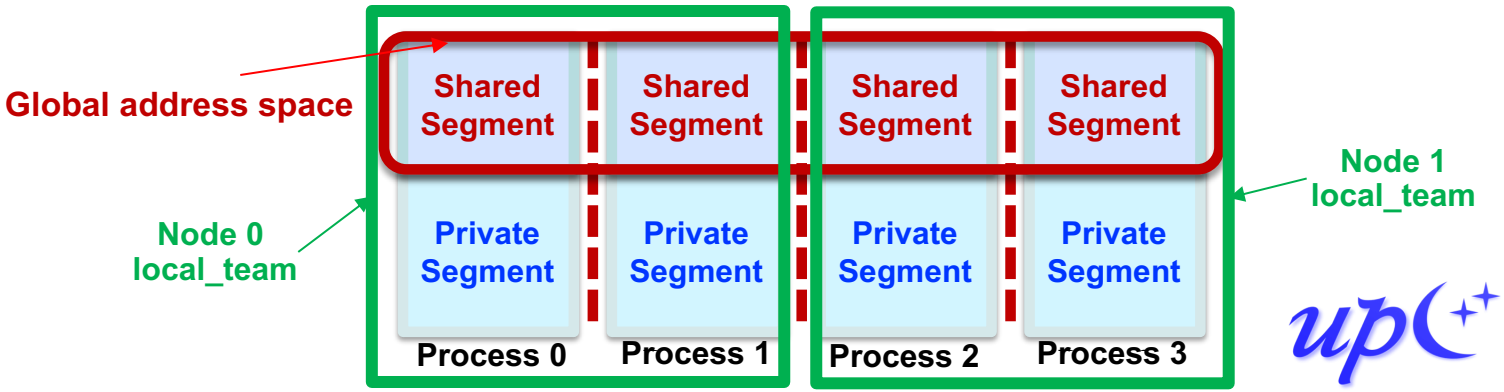
- Some process pairs are “closer” than others
- Ex: cabinet > switch > node > NUMA domain > socket > core

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 processes in the same `local_team`



Downcasting and shared-memory bypass

Earlier we covered downcasting global pointers

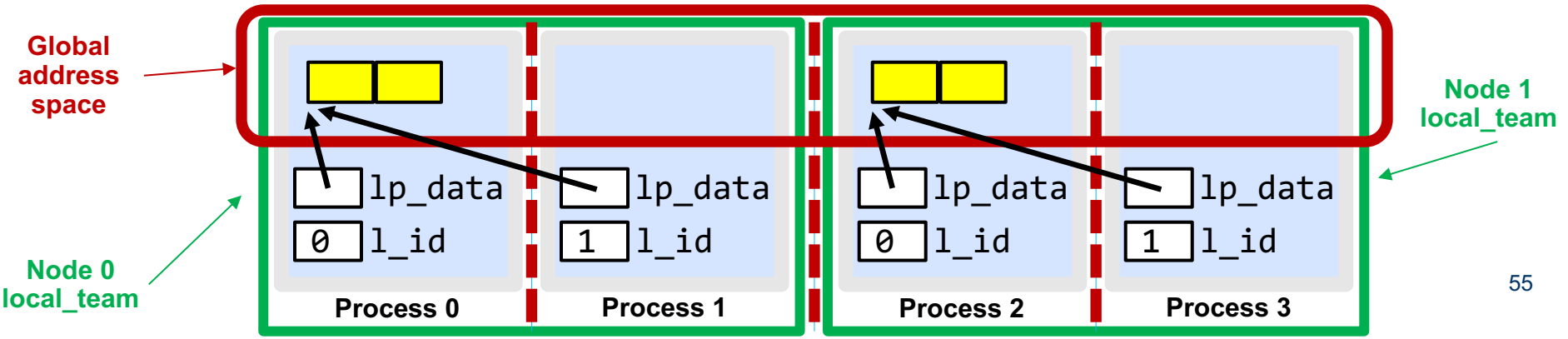
- Converting `global_ptr<T>` from this process to raw C++ `T*`
- Also works for `global_ptr<T>` from **any** process in `local_team()`

```
int l_id  = local_team().rank_me();
int l_cnt = local_team().rank_n();
```
Rank and count in my local node

```
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();
```
Allocate and share one array **per node**

```
int *lp_data = gp_data.local();
```
Downcast to get raw C++ ptr to shared array

```
lp_data[l_id] = l_id;
```
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:

```
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

```
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();
}
```

Close registration and obtain an associated future

Register an operation on a promise

Completion: "signaling put"

One particularly interesting case of completion:

```
rput(src_lptr, dest_gptr, count,
     remote_cx::as_rpc([=]() {
       // callback runs at target after put arrives
       compute(dest_gptr, 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

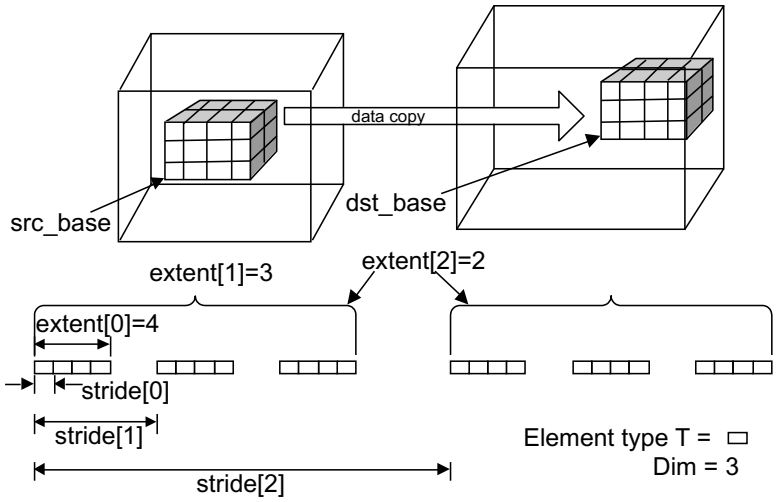
Supercomputers are becoming increasingly heterogeneous in compute, memory, storage

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

```
global_ptr<int, memory_kind::cuda_device> src = ...;
global_ptr<int, memory_kind::cuda_device> dst = ...;

copy(src, dst, N).wait();
```

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

UPC++ additional resources

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

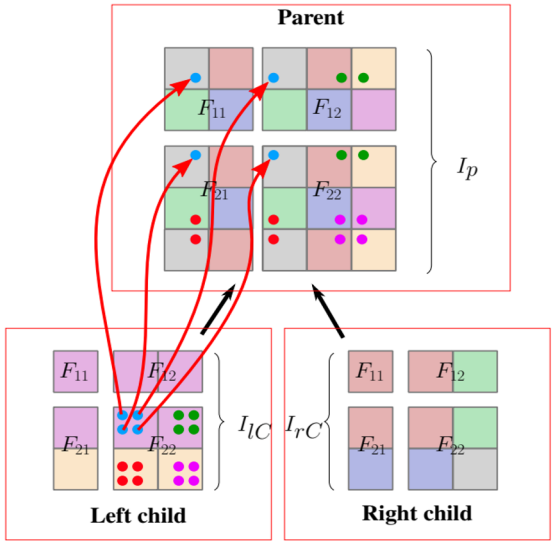
- Open-source/free library implementation
  - Portable from laptops to supercomputers
- Tutorial resources at **upcxx.lbl.gov/training**
  - UPC++ Programmer's Guide
  - Videos and exercises from past tutorials
- Formal UPC++ specification
  - All the semantic details about all the features
- Links to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

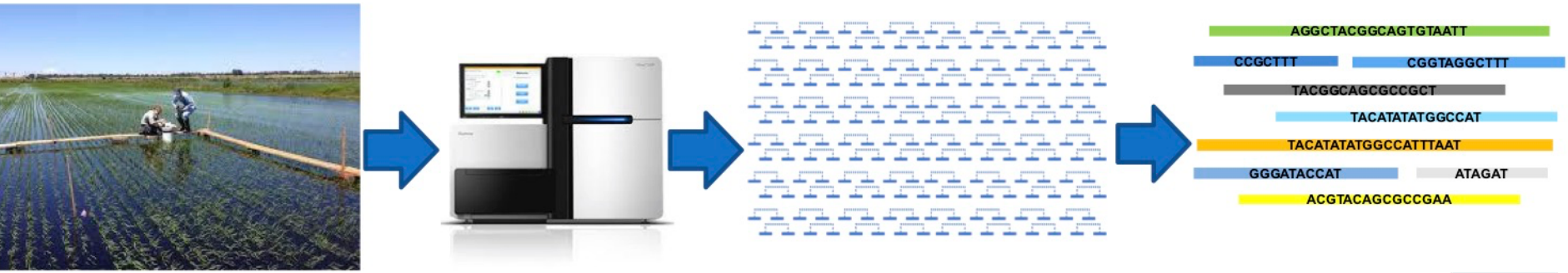
Application case studies

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

We discuss two specific applications:

- symPack, a solver for sparse symmetric matrices
- MetaHipMer, a genome assembler





Sparse multifrontal direct linear solver

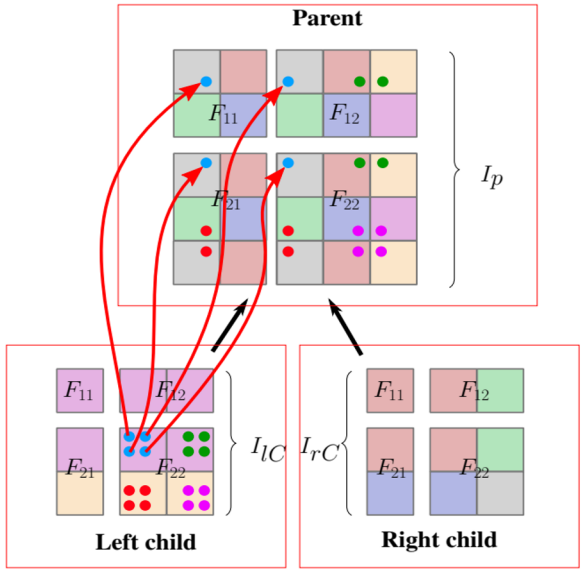
Sparse matrix factorizations have low computational intensity and irregular communication patterns

Extend-add operation is an important building block for **multifrontal sparse solvers**

Sparse factors are organized as a hierarchy of condensed matrices called **frontal matrices**

Four sub-matrices:
factors + contribution block

Code available as part of upcxx-extras BitBucket git repo



Details in IPDPS'19 paper:

Bachan, Baden, Hofmeyr, Jacquelin, Kamil, Bonachea, Hargrove, Ahmed. "UPC++: A High-Performance Communication Framework for Asynchronous Computation", https://doi.org/10.25344/S4V88H

Implementation of the extend-add operation

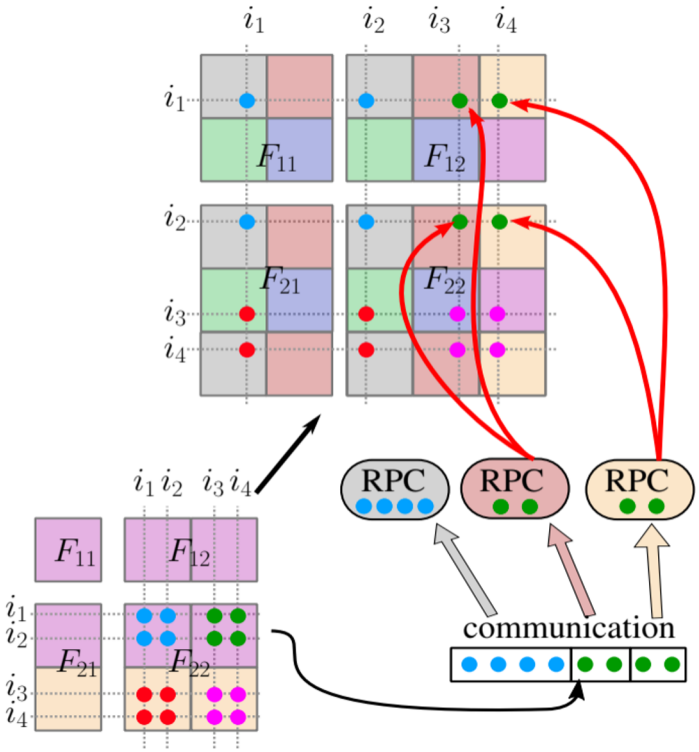
Data is binned into per-destination contiguous buffers

Traditional MPI implementation uses `MPI_Alltoallv`

- Variants: `MPI_Isend`/`MPI_Irecv` + `MPI_Waitall`/`MPI_Waitany`

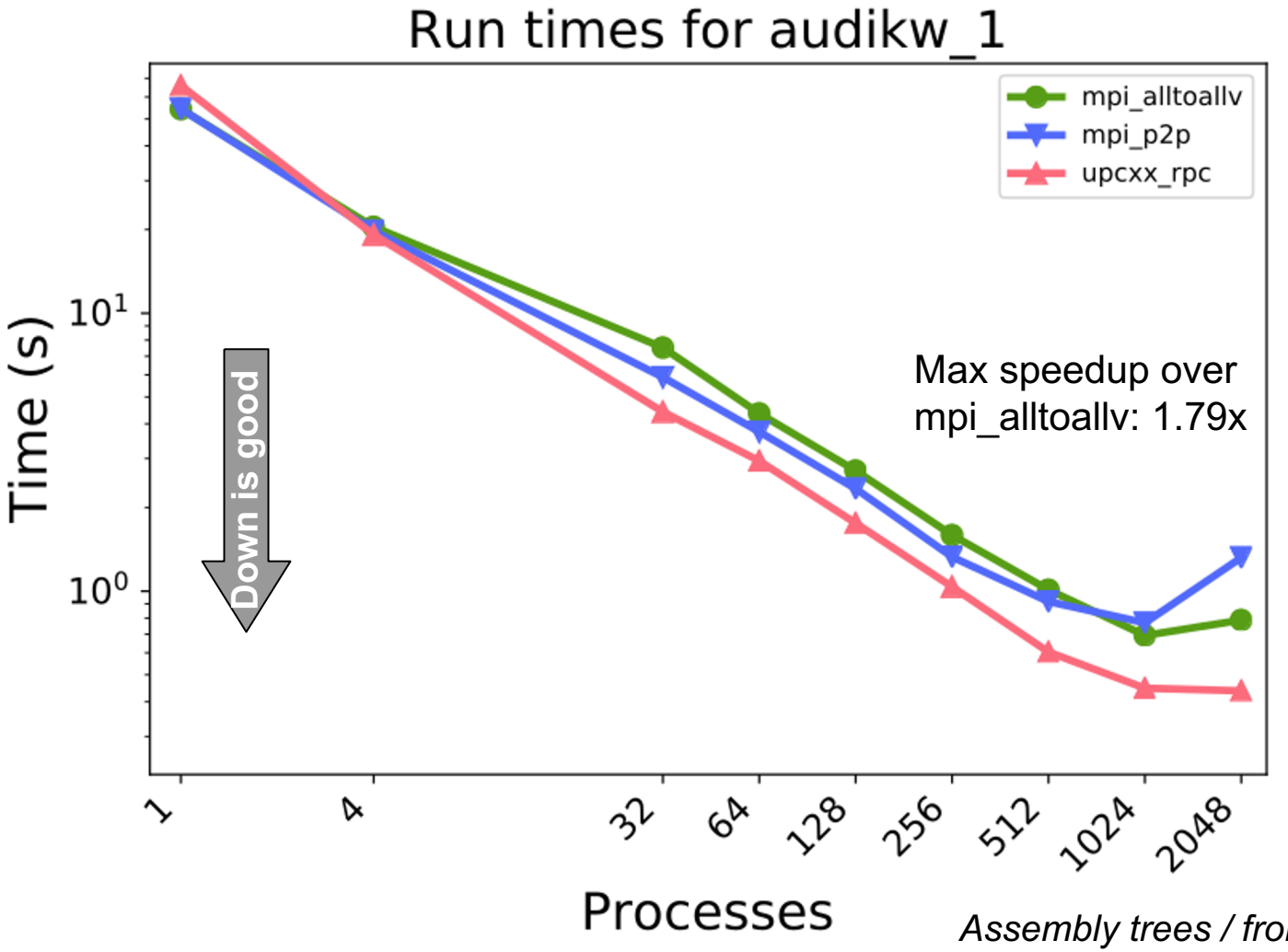
UPC++ Implementation:

- RPC sends child contributions to the parent using a UPC++ ***view***
- RPC callback compares indices and accumulates contributions on the target



Details in IPDPS'19 https://doi.org/10.25344/S4V88H

UPC++ improves sparse solver performance
Experiments done on Cori Haswell

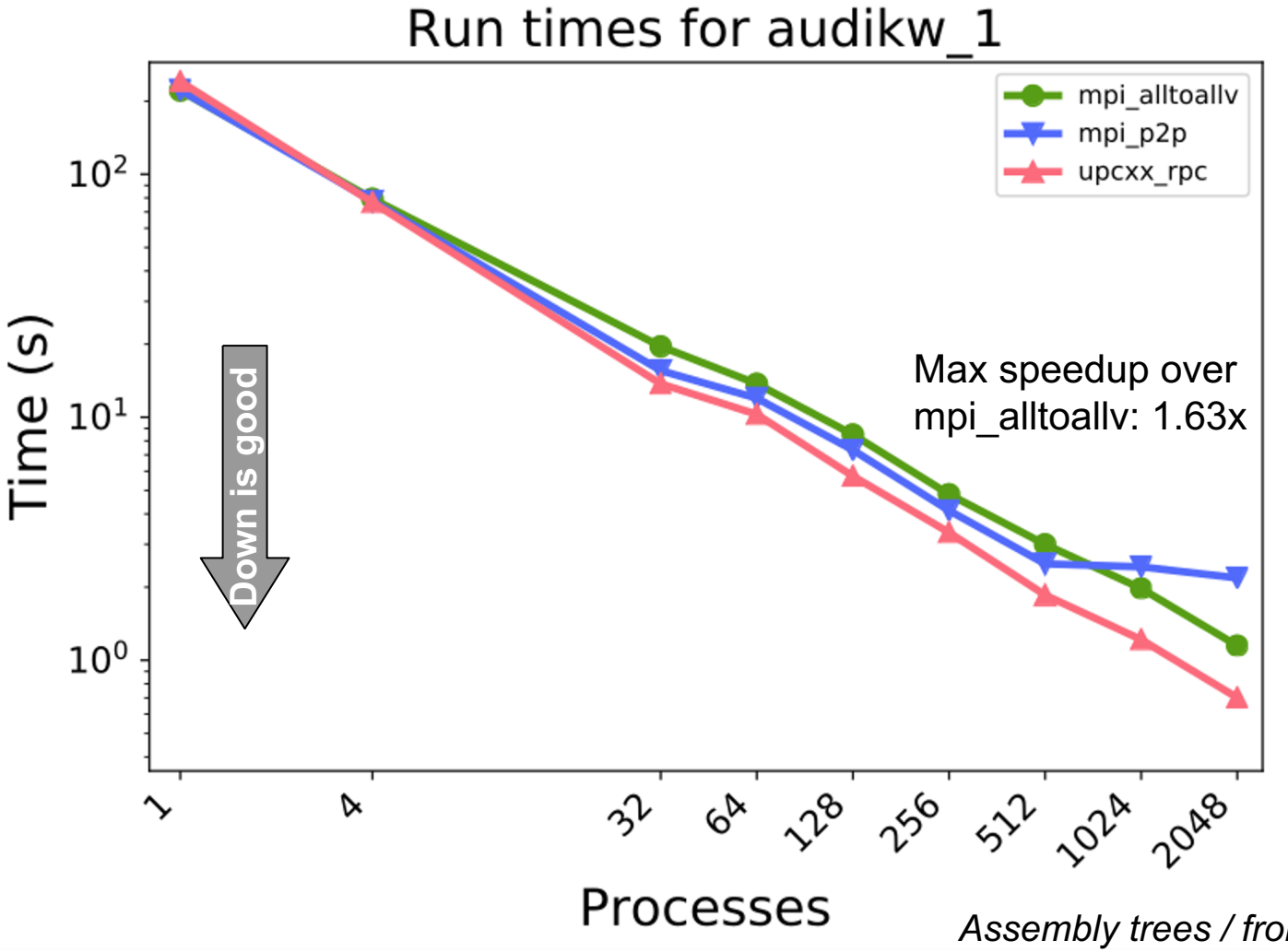


Assembly trees / frontal matrices extracted from STRUMPACK

Details in IPDPS'19 https://doi.org/10.25344/S4V88H

UPC++ improves sparse solver performance

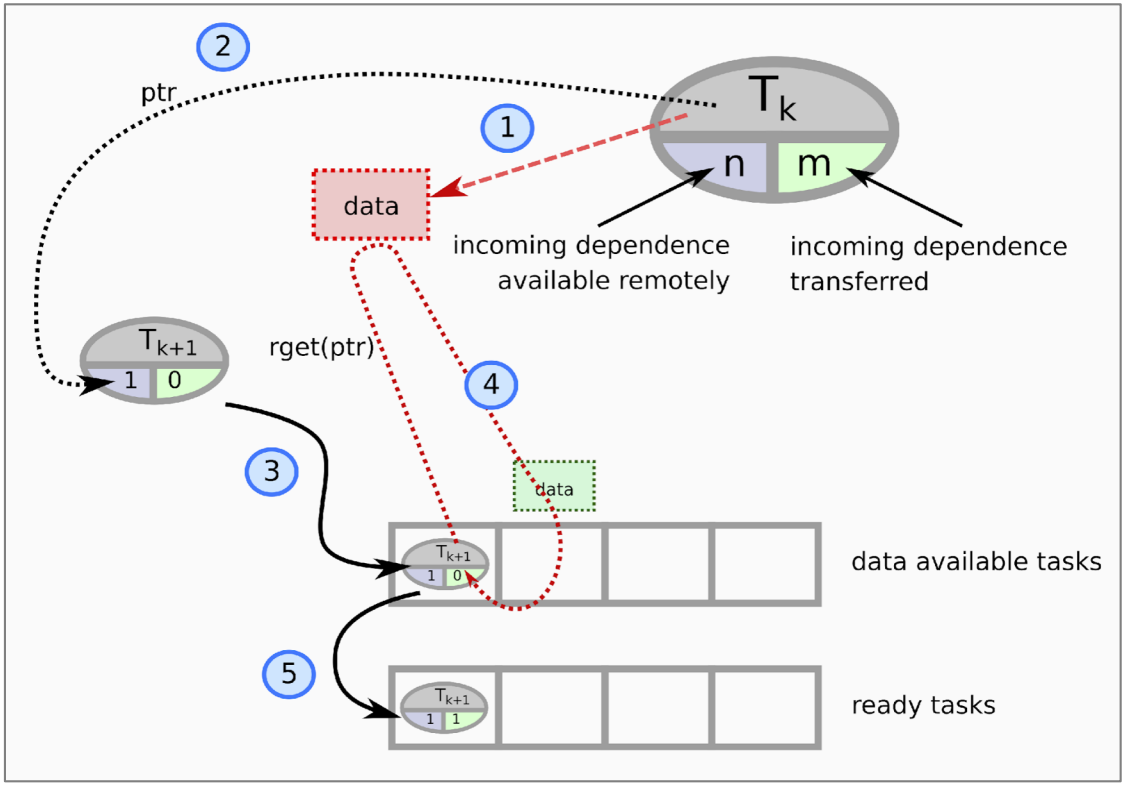
Experiments done on Cori KNL



Assembly trees / frontal matrices extracted from STRUMPACK

Details in IPDPS'19 https://doi.org/10.25344/S4V88H

symPACK: a solver for sparse symmetric matrices



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

symPACK: a solver for sparse symmetric matrices

Matrix is distributed by supernodes

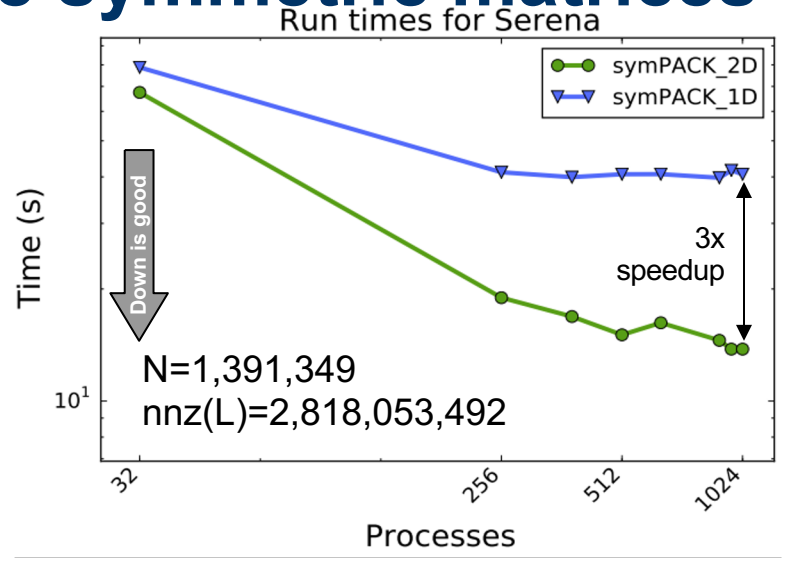
- 1D distribution
  - Balances flops, memory
  - Lacks strong scalability
- New 2D distribution (to appear)
  - Explicit load balancing, not regular block cyclic mapping
  - Balances flops, memory
  - Finer granularity task graph

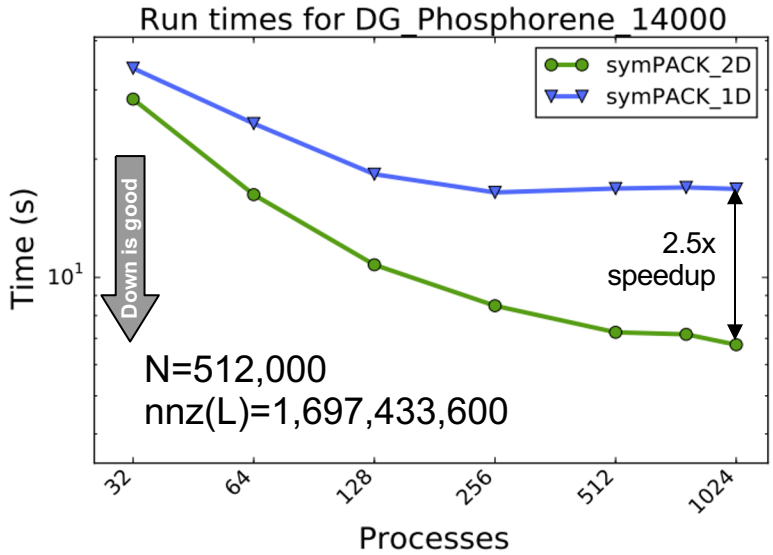
Strong scalability on Cori Haswell:

- 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

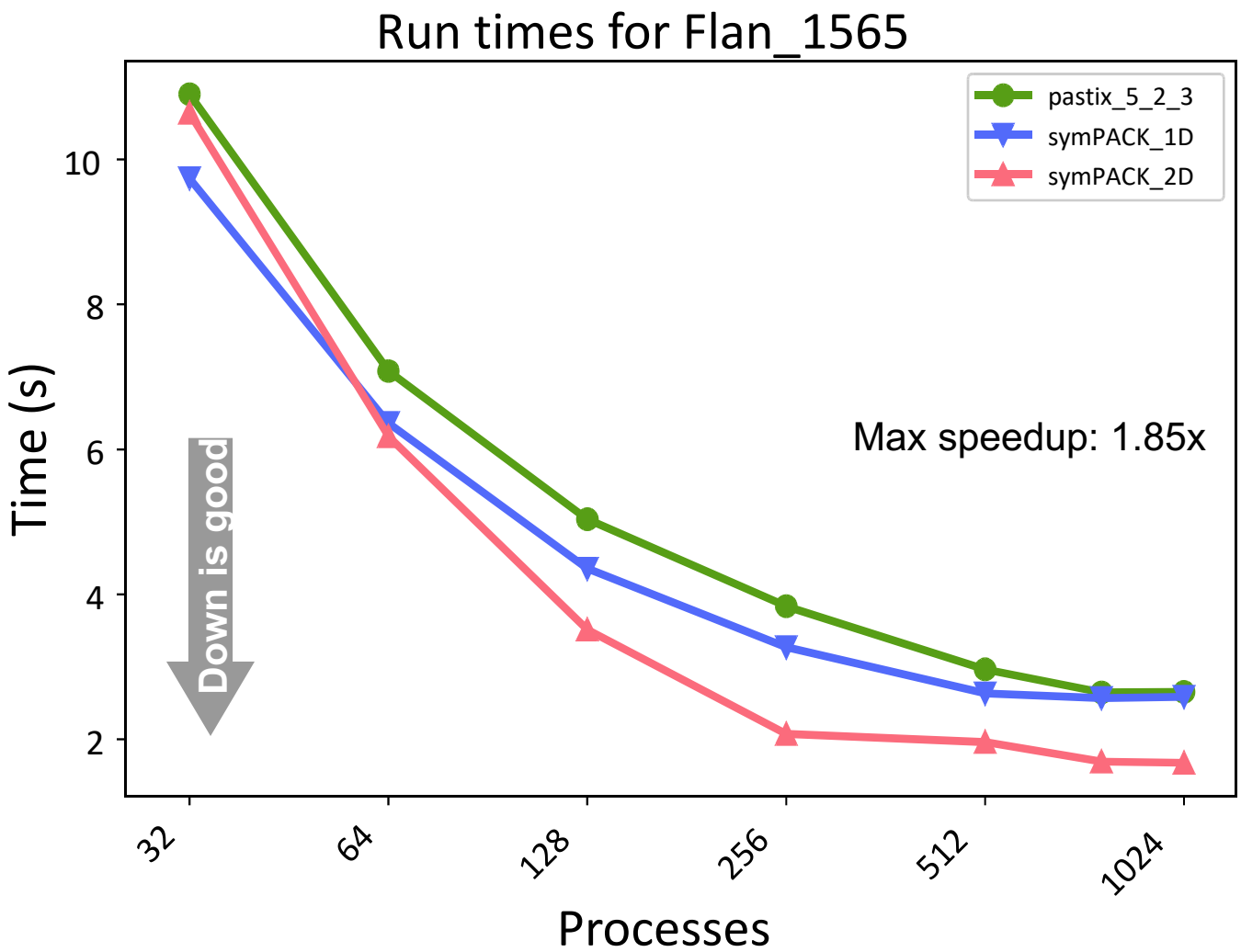
- **Better strong scalability**





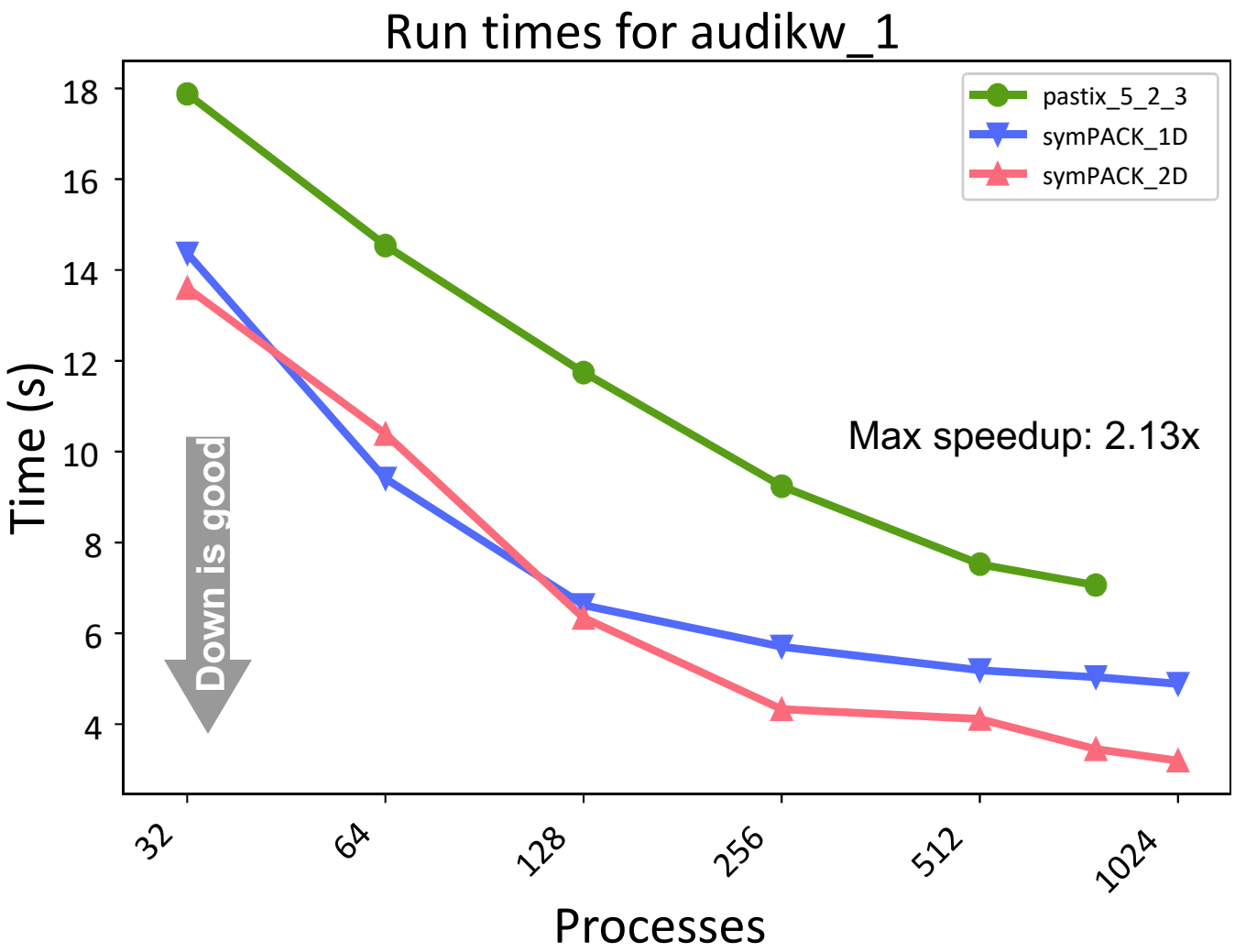
Work and results by Mathias Jacquelin, funded by SciDAC CompCat and FASTMath

symPACK strong scaling experiment
NERSC Cori Haswell



N=1,564,794 nnz(L)=1,574,541,576

symPACK strong scaling experiment
NERSC Cori Haswell



N=943,695 nnz(L)=1,261,342,196

UPC++ provides productivity + performance for sparse solvers

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
- Increased efficiency in the extend-add operation
- Outperform state-of-the-art sparse symmetric solvers

http://upcxx.lbl.gov

http://sympack.org

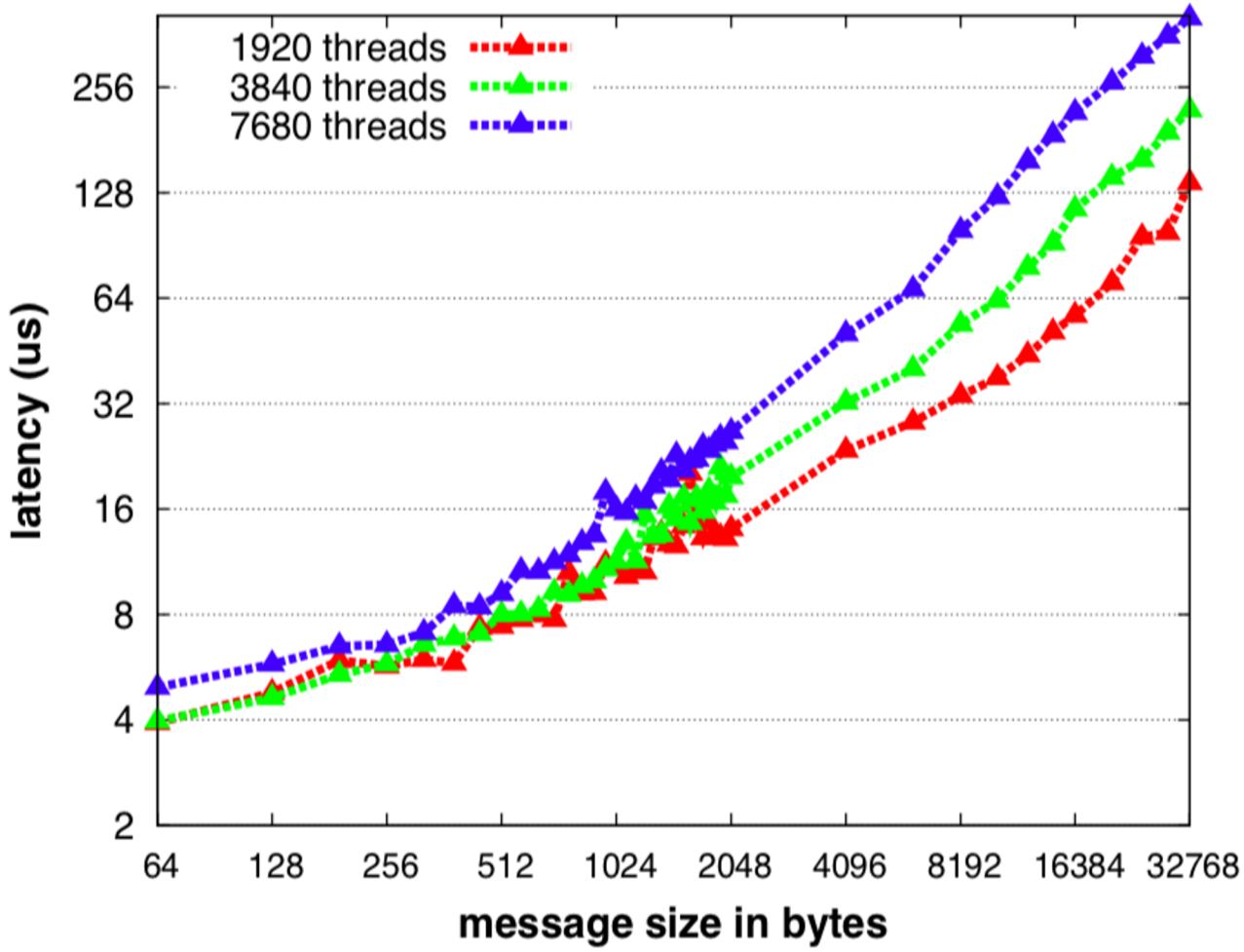
ExaBiome / MetaHipMer distributed hashmap

Memory-limited graph stages

- k-mers, contig, scaffolding

Optimized graph construction

- Larger messages for better network bandwidth



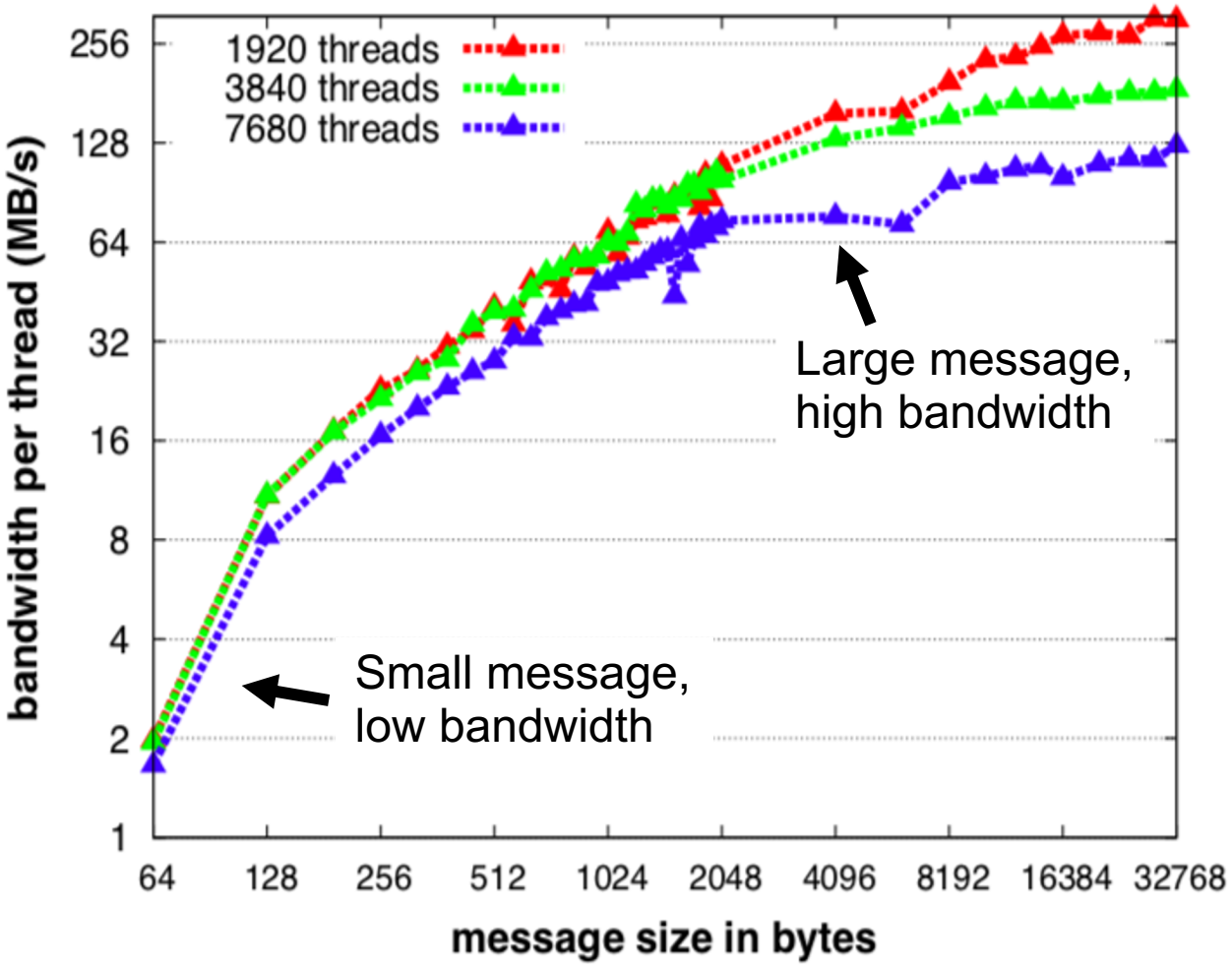
ExaBiome / MetaHipMer distributed hashmap

Memory-limited graph stages

- k-mers, contig, scaffolding

Optimized graph construction

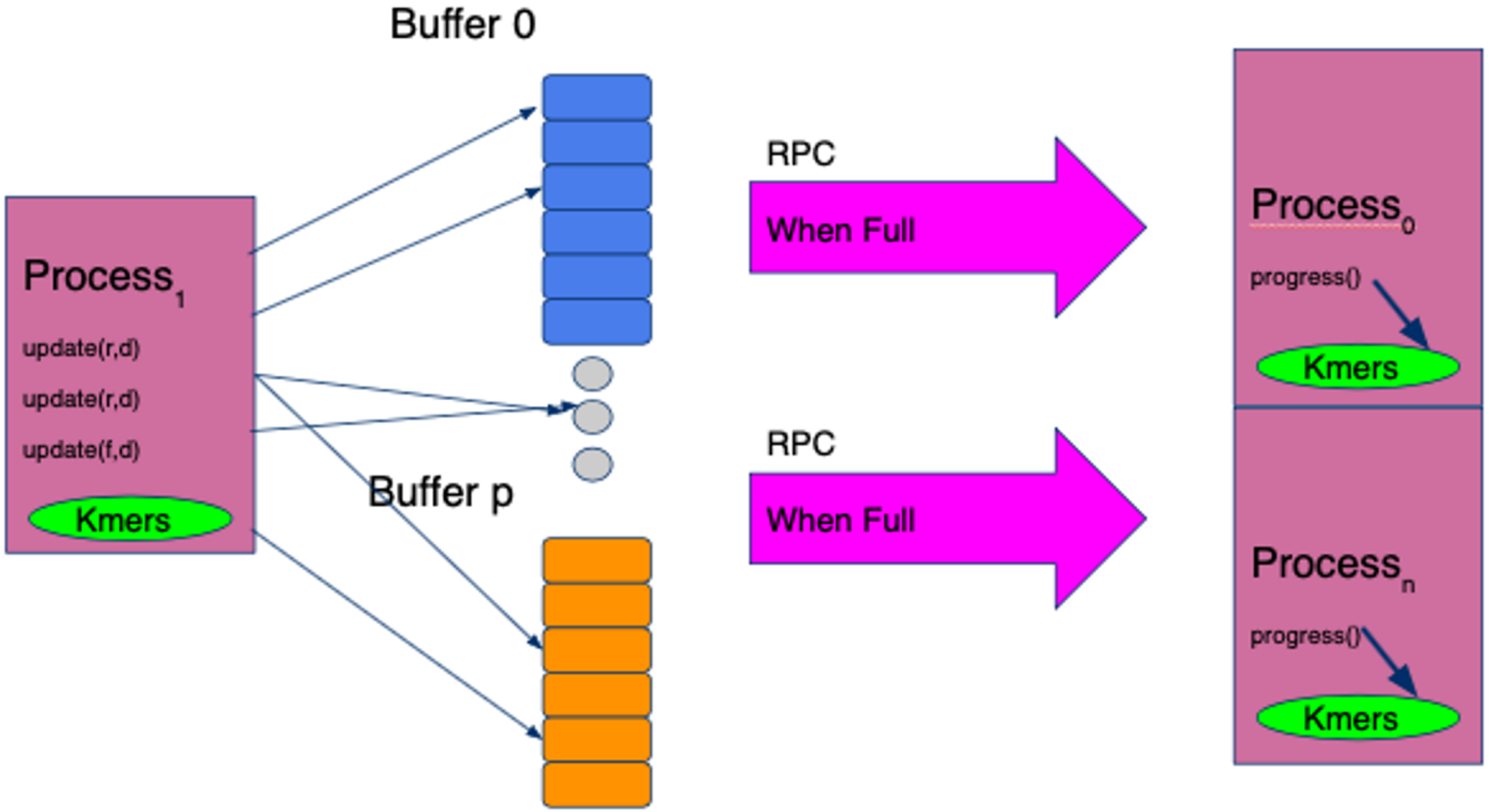
- Larger messages for better network bandwidth



ExaBiome / MetaHipMer distributed hashmap

Aggregated store

- Buffer calls to `dist_hash::update(key,value)`
- Send fewer but larger messages to target rank



API - AggrStore<FuncDistObject, T>

```
struct FunctionObject {
  void operator()(T &elem) { /* do something */ }
};
using FuncDistObject = upcxx::dist_object<FunctionObject>;

// AggrStore holds a reference to func
AggrStore(FuncDistObj &func);
~AggrStore() { clear(); }

// clear all internal memory
void clear();

// allocate all internal memory for buffering
void set_size(size_t max_bytes);

// add one element to the AggrStore
void update(intrank_t target_rank, T &elem);

// flush and quiesse
void flush_updates();
```

MetaHipMer utilized UPC++ features

C++ templates - efficient code reuse

`dist_object` - as a templated functor & data store

Asynchronous all-to-all exchange - not batch sync

- 5x improvement at scale over previous MPI implementation

Future-chained workflow

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

Promise & fulfill - for a fixed-size memory footprint

- Issue promise when full, fulfill when available

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 for support