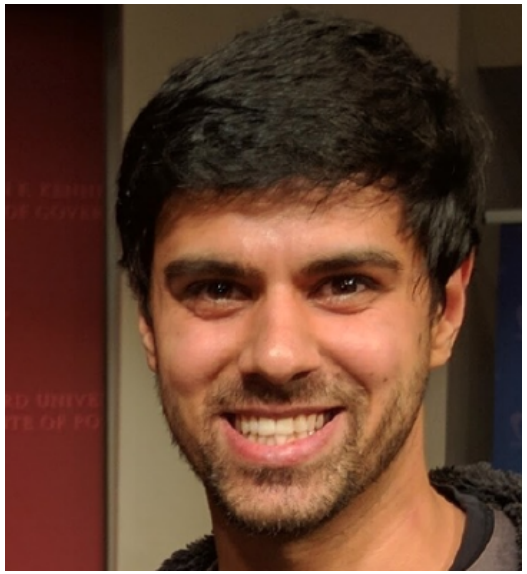
Special Lecture on Thursday: 2/25

Wonderful GSIs will talk about their research and related project ideas!

Giulia Guidi



Melih Elibol

Alok Tripathy

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

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

Kathy Yelick, Amir Kamil
Dan Bonachea, Paul H. Hargrove

https://upcxx.lbl.gov/sc20
pagoda@lbl.gov

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 is a condensed version of a tutorial presented at SC20. The examples, downloads, longer videos by the team who design UPC++ and more is available here: **https://upcxx.lbl.gov/sc20**

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 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..

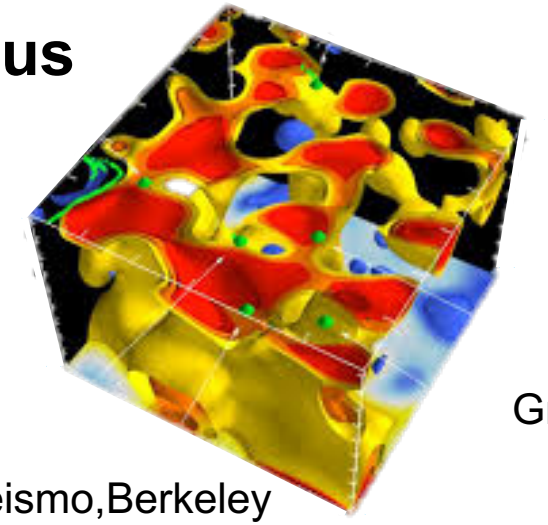
Some motivating applications

Many applications involve asynchronous updates to irregular data structures

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

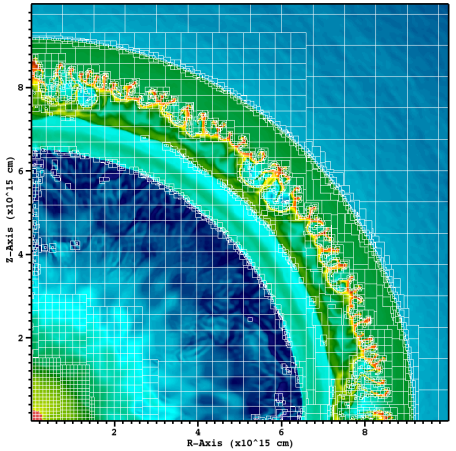
Irregular and unpredictable data movement:

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

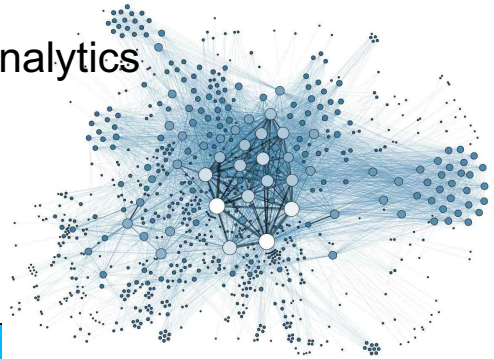


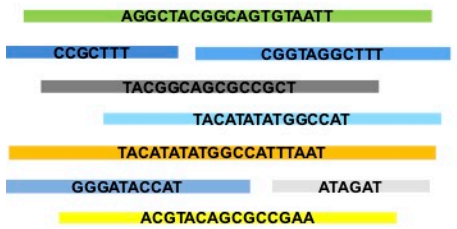
Seismo,Berkeley

AMReX

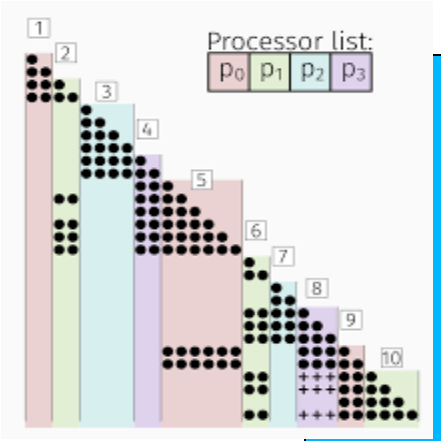


Graph analytics





ExaBiome

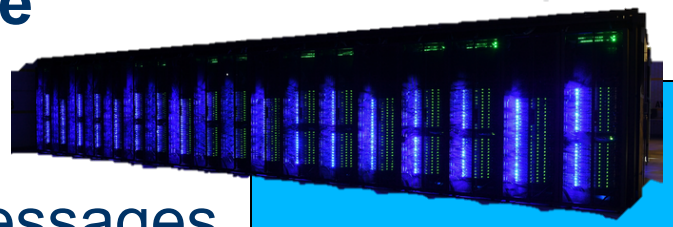


SymPACK

Some motivating system trends

The first exascale systems will appear in 2021

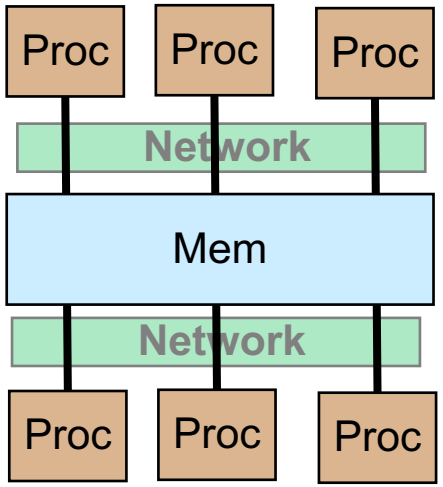
- Cores per node is growing
- Cores are getting simpler (including GPU cores)
- Memory per core is dropping
- Latency is not improving

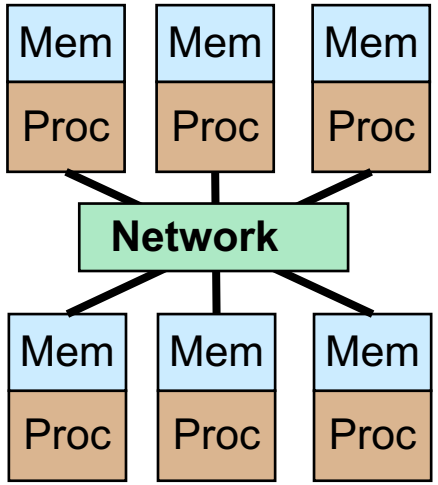


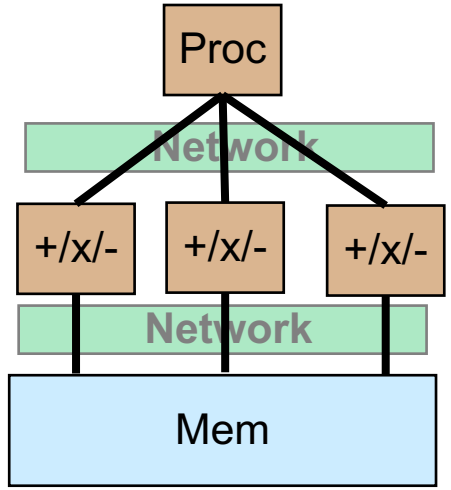
Need to reduce communication costs in software

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

Parallel Machines and Programming



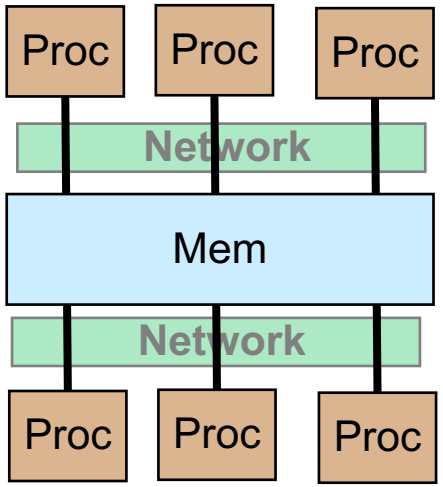


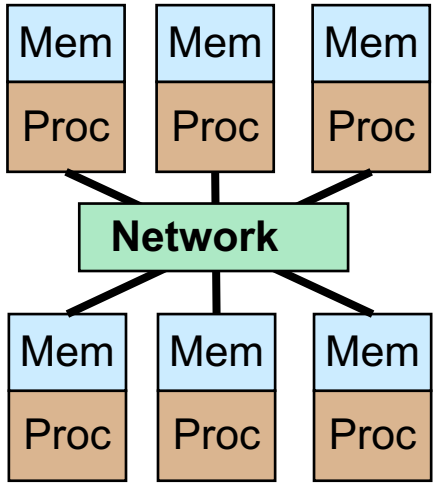


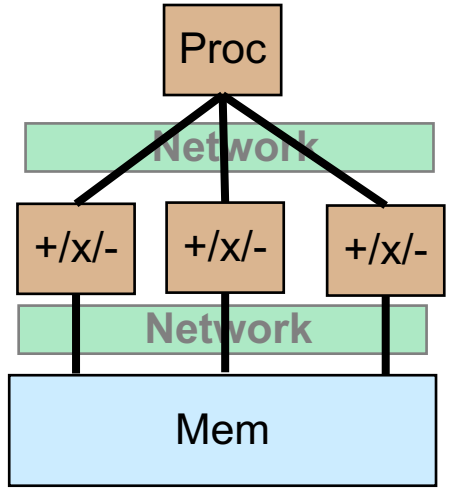
Shared Memory	Distributed Memory	Single Instruction Multiple Data (SIMD)
OpenMP, Threads	**MPI (send/receive)**	**Data parallel (collectives)**
Processors execute own instruction stream	Processors execute own instruction stream	One instruction stream (all run same instruction)
Communicate by reading/writing memory	Communicate by sending messages	Communicate through memory
Cost of a read/write is constant	Message time depends on size, but not location	Assume unbounded # of arithmetic units

Ideal cost

Parallel Machines and Programming







Key PGAS "feature": never cache remote data

Shared Memory	Distributed Memory	Single Instruction Multiple Data (SIMD)
OpenMP, Threads	**MPI (send/receive)**	**Data parallel (collectives)**
Processors execute own instruction stream	Processors execute own instruction stream	One instruction stream (all run same instruction)
Communicate by reading/writing memory	Communicate by sending messages	Communicate through memory
Cost of a read/write is constant	Memory access time depends on size and whether local vs. remote	Assume unbounded # of arithmetic units

Ideal cost

upC++ BERKELEY LAB

Advantages and disadvantages of each

Shared memory / OpenMP

+**Ease:** Easier to parallelize existing serial code

Correctness: Race conditions

Scalability: No locality control; cache coherence doesn't scale

Performance: False sharing, lack of parallelism, etc.

Message Passing / two-sided MPI

Ease: More work up front to partition data

+**Correctness:** Harder to create races (although deadlocks can still be a problem)

+**Scalability**: Effectively unlimited

+**Performance:** More transparent, but messages are expensive (need to pack/unpack)

Parallel Programming Problem: Histogram

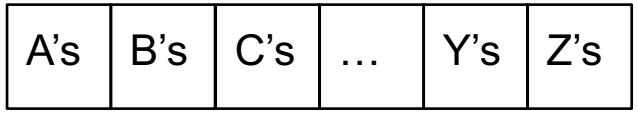
Consider the problem of computing a histogram:

Large number of “words” streaming in from somewhere

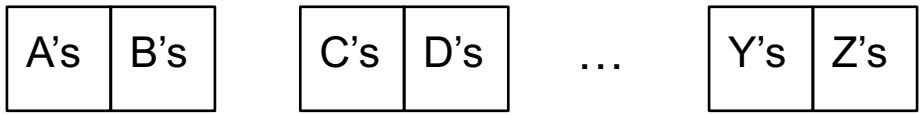
You want to count the # of words with a given property

Shared memory

Lock each bucket

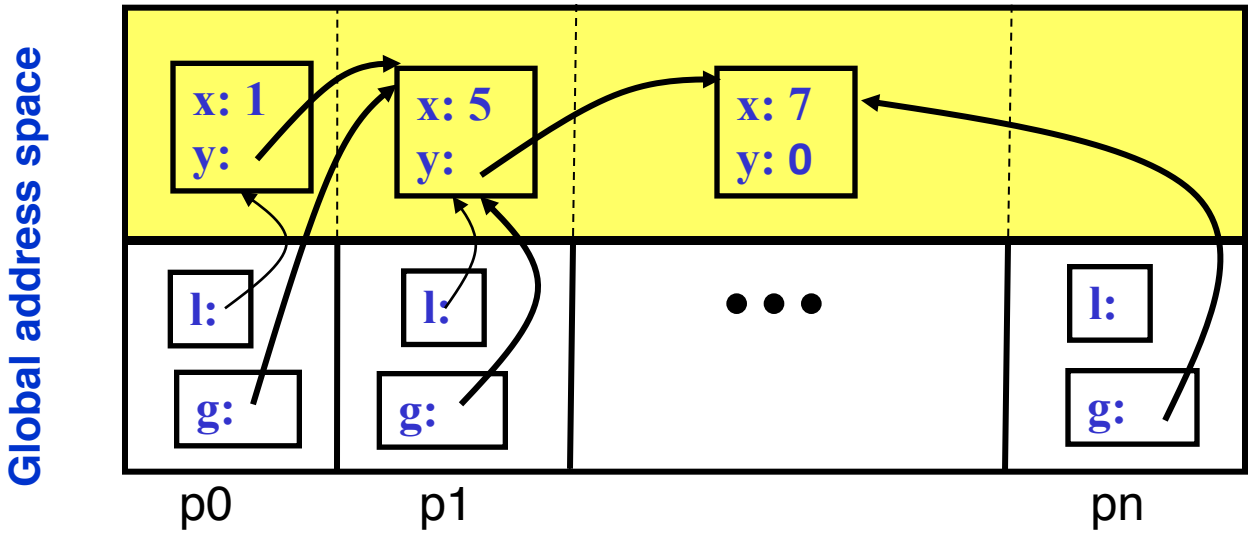


- Message passing: the array is huge and spread out
  - Each processor has a substream and sends +1 to the appropriate processor…
  - When does that processor “receive”?

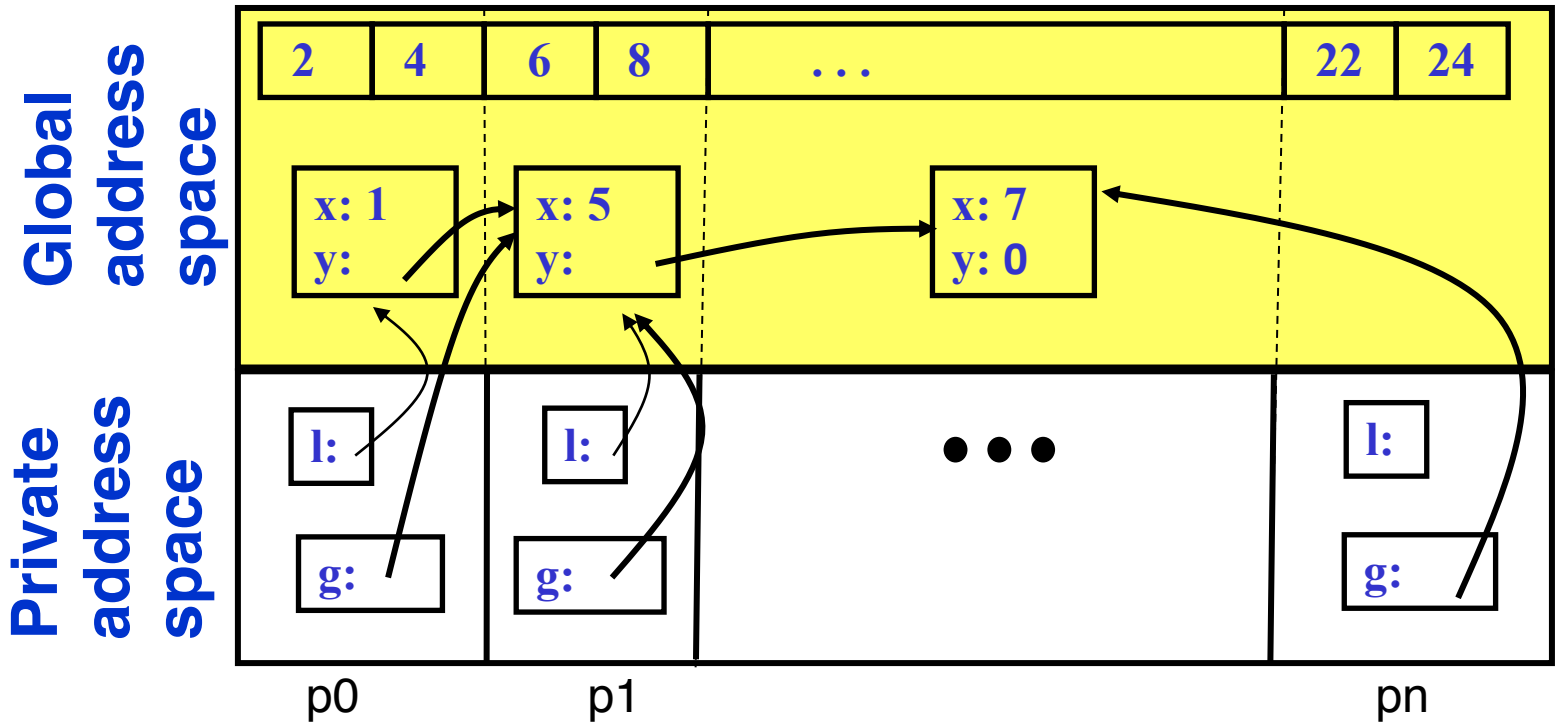


PGAS = Partitioned Global Address Space

- *Global address space:* thread may directly read/write remote data
  - Convenience of shared memory
- *Partitioned:* data is designated as local or global
  - Locality and scalability of message passing



Partitioned Global Address Space (PGAS)

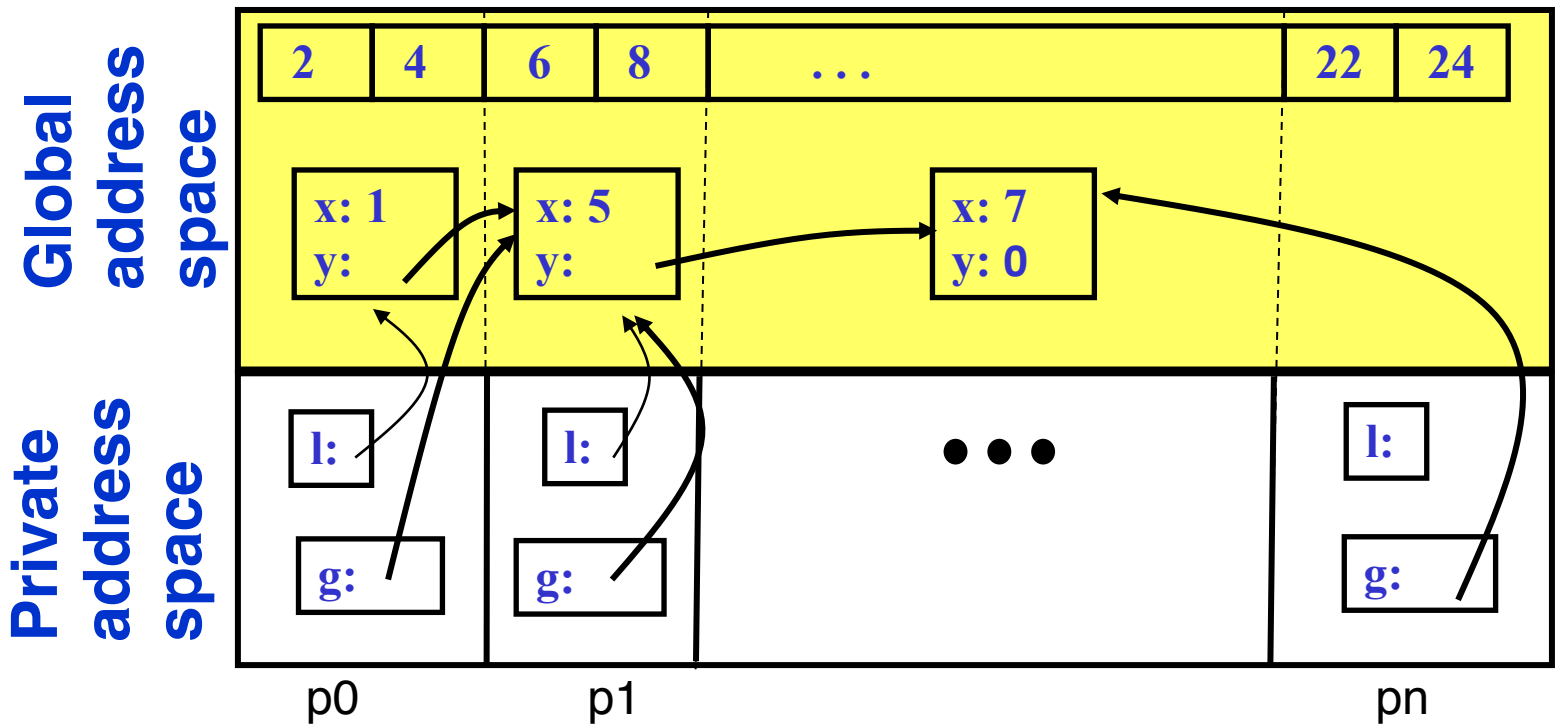


Need a way to name remote memory (UPC syntax)

Global pointers: shared int * p = upc_malloc(4);

Distributed arrays: shared int a [12];

One-sided communication in PGAS



Directly read/write remote memory; partitioned for locality

One-sided communication underneath (UPC syntax):

Put: a[i] = … ; *p = ...; upc_mem_put(..)

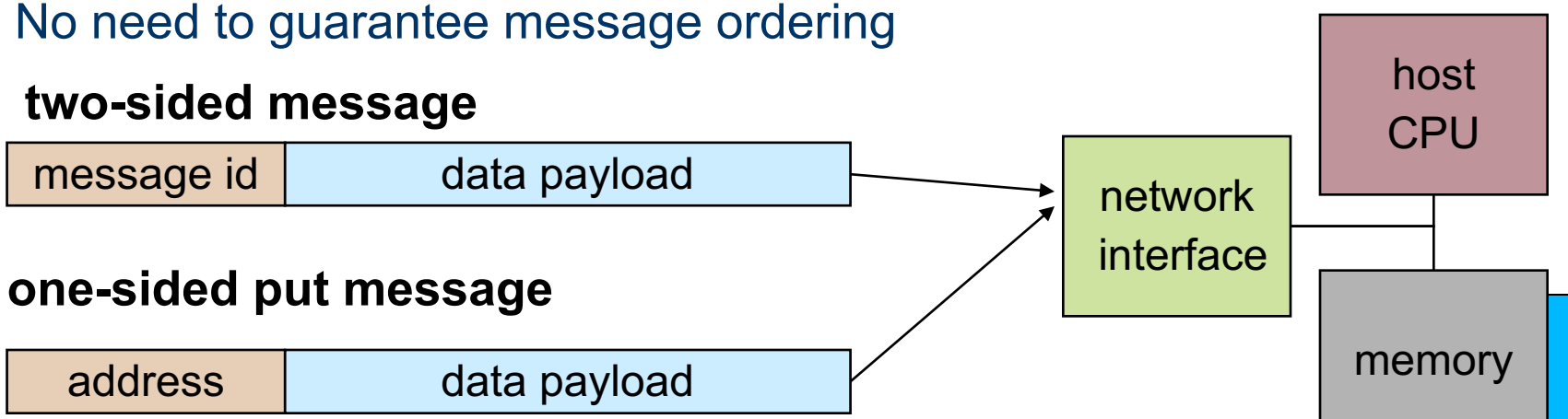
Get: ... = a[i]...; ... = *p; upc_mem_get(...)

Reducing communication overhead

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

Communication is **one-sided**

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

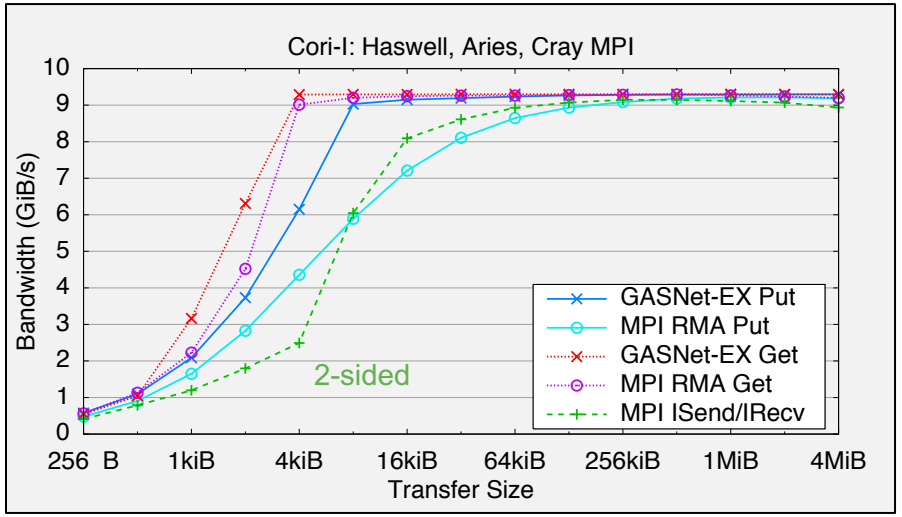


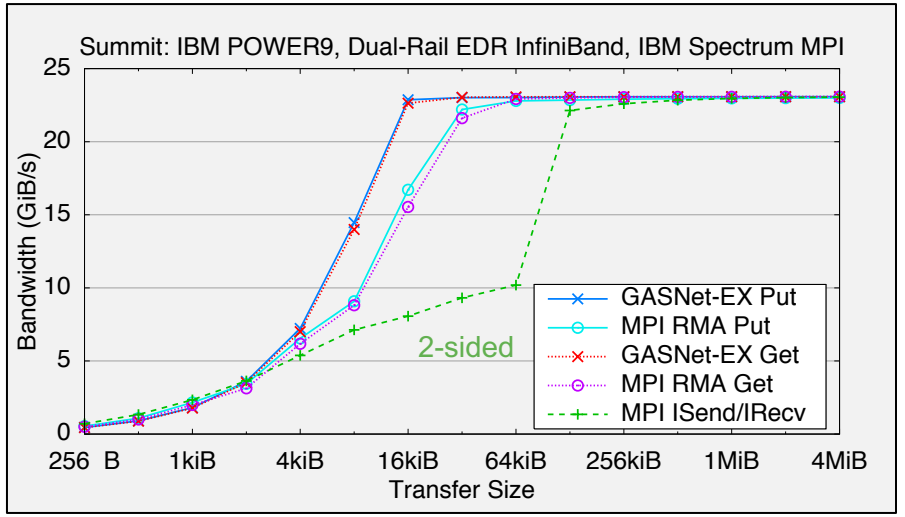
- Metadata from sender, rather than split between sender / receiver
- Supported in hardware through RDMA (Remote Direct Memory Access)

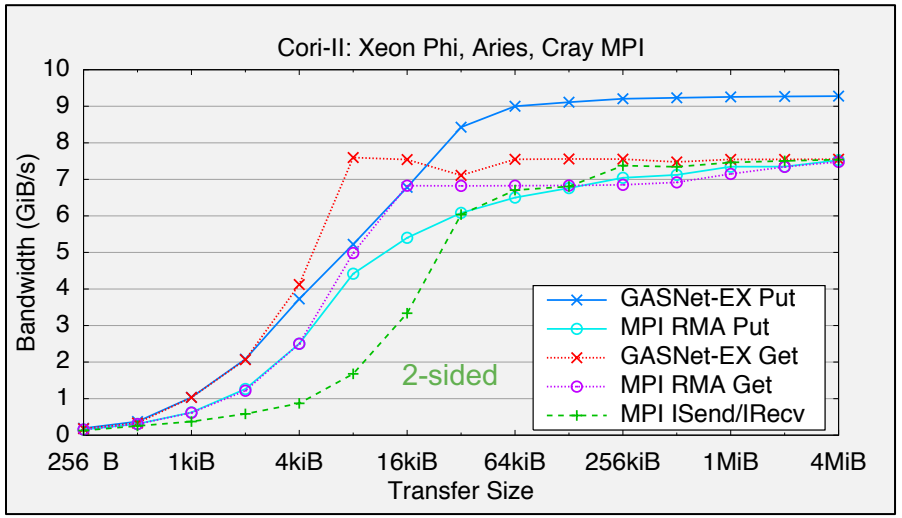
Like shared memory: shared data structures with asynchronous access

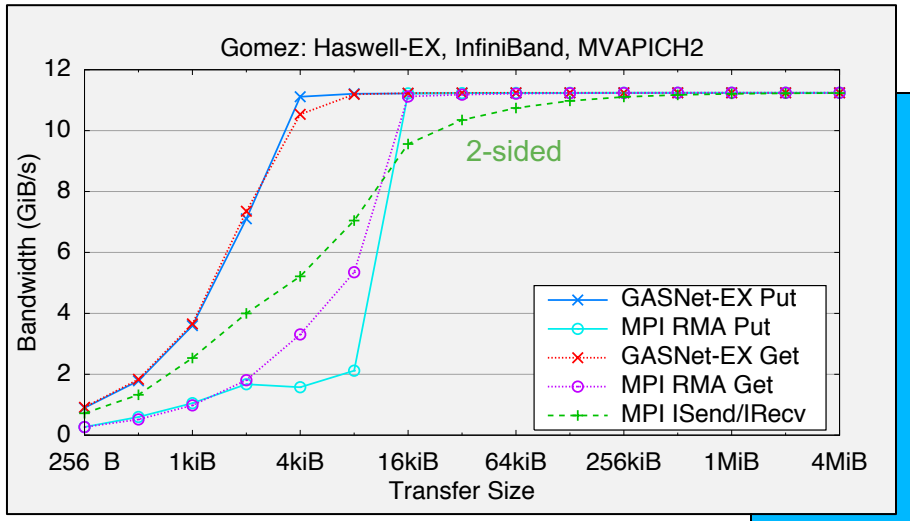
One-sided vs Two-sided Message Performance

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





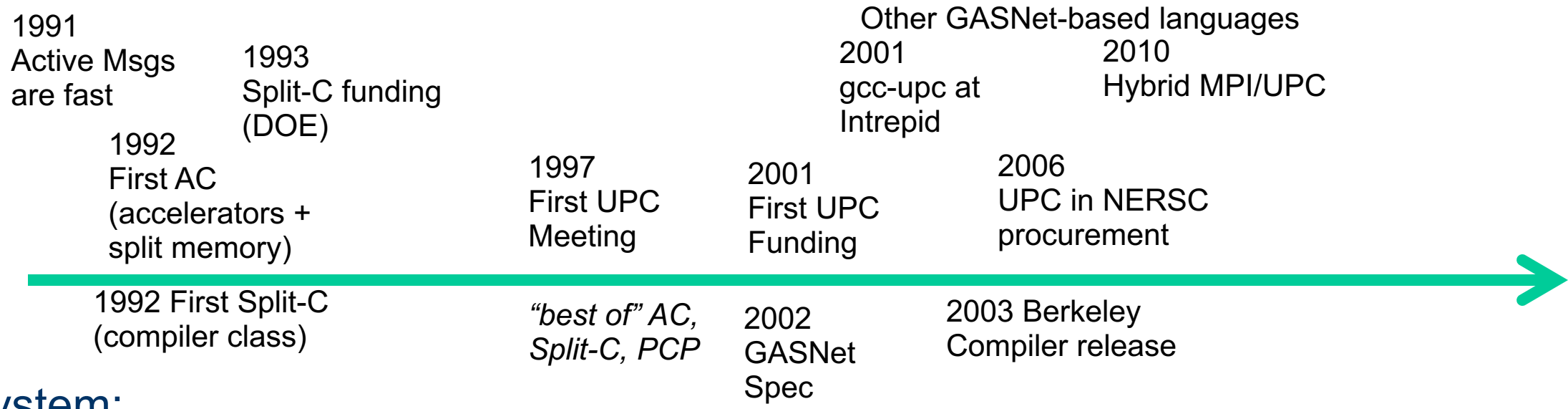




UP IS GOOD

- MPI ISend/IRecv is 2-sided
- All others are 1-sided

PGAS History at Berkeley



Ecosystem:

- Users with a need (fine-grained random access)
- Machines with RDMA (not full cache-coherence)
- Common runtime; Commercial and free software

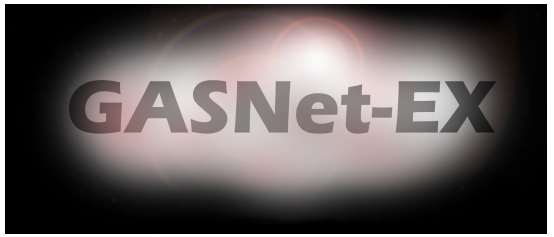
Languages and Libraries:

- UPC, Co-Array Fortran, Titanium, Chapel, X10, UPC++…
- Libraries : Habanero UPC++, OpenSHMEM, Co-Array C++, Global Arrays, DASH, UPC++
- Influence traditional models: MPI 1-sided; OpenMP locality control

UPC++: A Programming Library for PGAS

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 more UPC++ RPCs are built
- Enables portability (laptops to supercomputers)

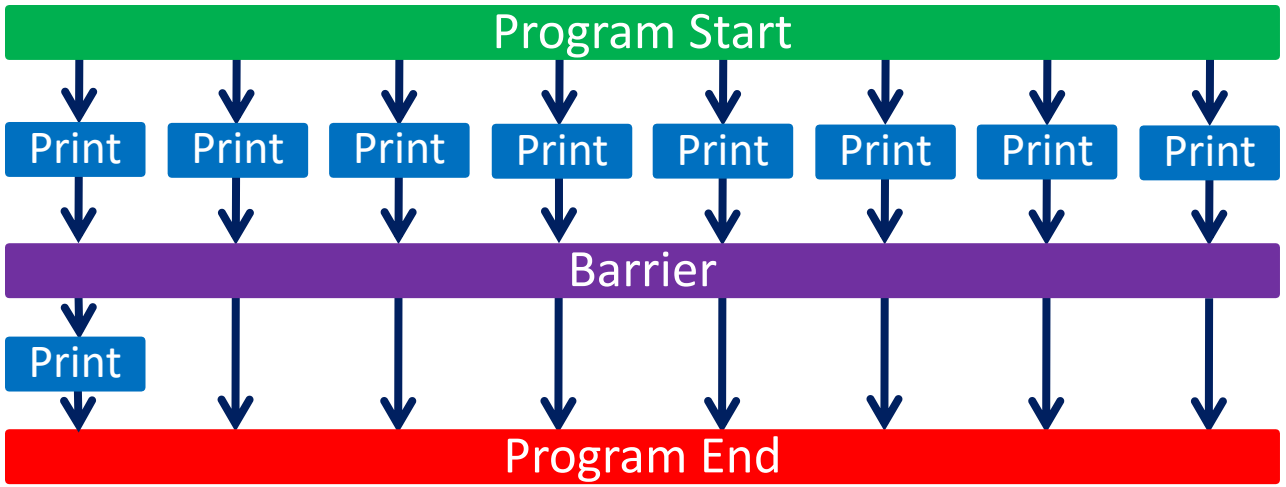
Designed for interoperability

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

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();
}
```



Compiling and running a UPC++ program

NERSC's Cori

```
$ module load upcxx
```

Compiler wrapper:

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

- Invokes a normal backend C++ compiler with the appropriate arguments (`-I`/`-L` etc).
- There are other mechanisms for compiling: `upcxx-meta` and CMake package

Launch wrapper:

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

- Arguments similar to other familiar tools
- Also supports platform-specific tools: **`srun`**, **`jsrun`** and **`aprun`**

UPC++ works on laptops, clusters, within a docker container

https://upcxx.lbl.gov/ contains links to source, etc.

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

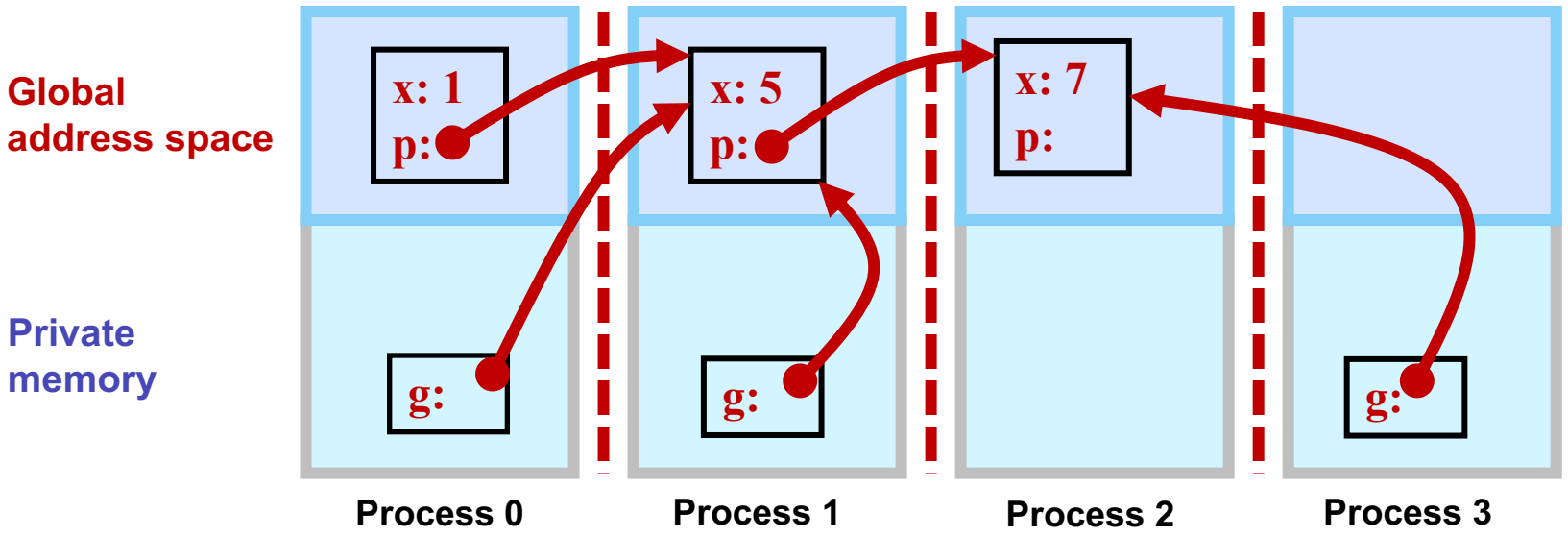
Close down UPC++ runtime

```
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
```

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>`

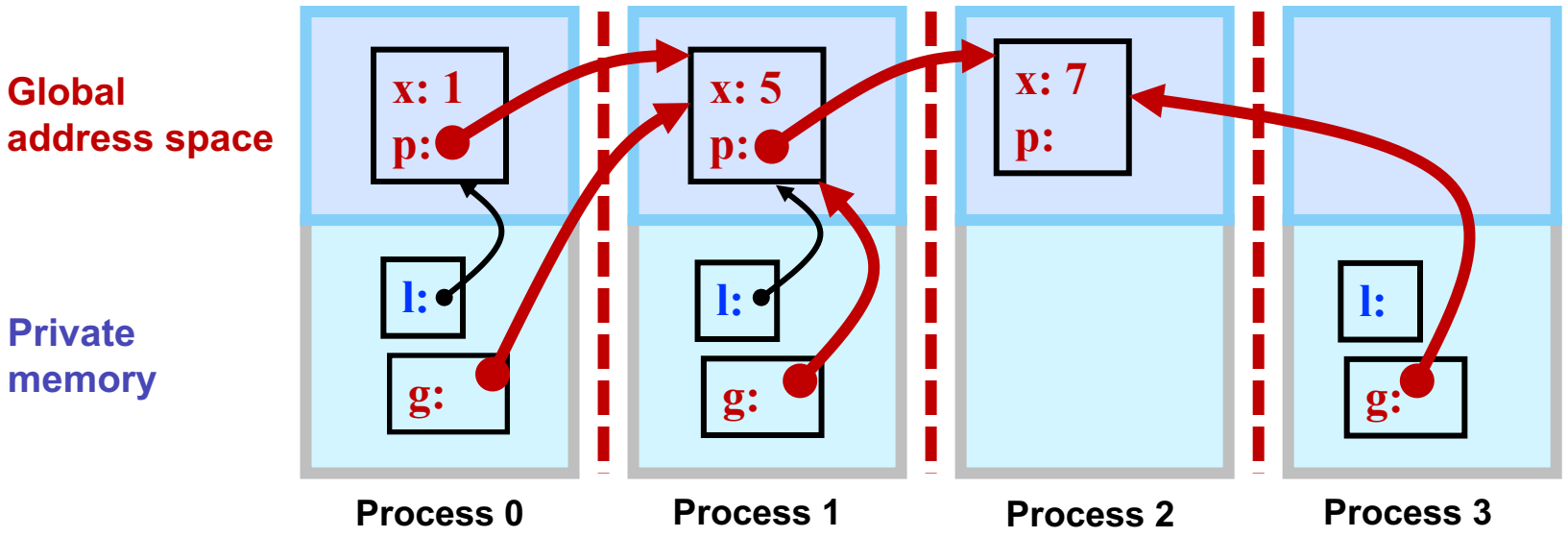


Global vs raw pointers and affinity

The affinity identifies the process that created the object

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

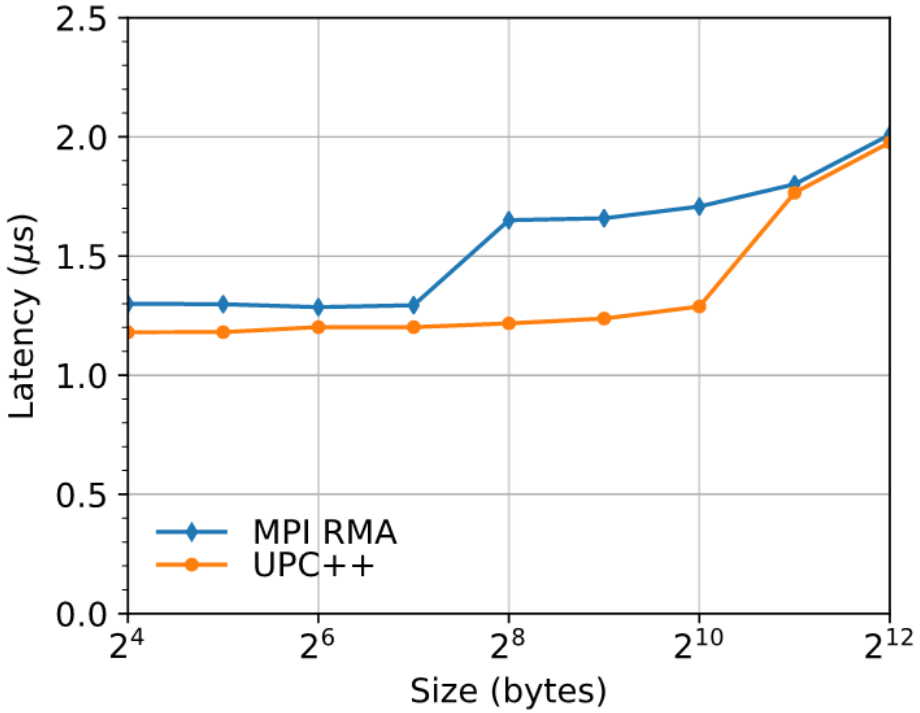
Raw C++ pointers can be used on a process to refer to objects in the global address space that have affinity to that process

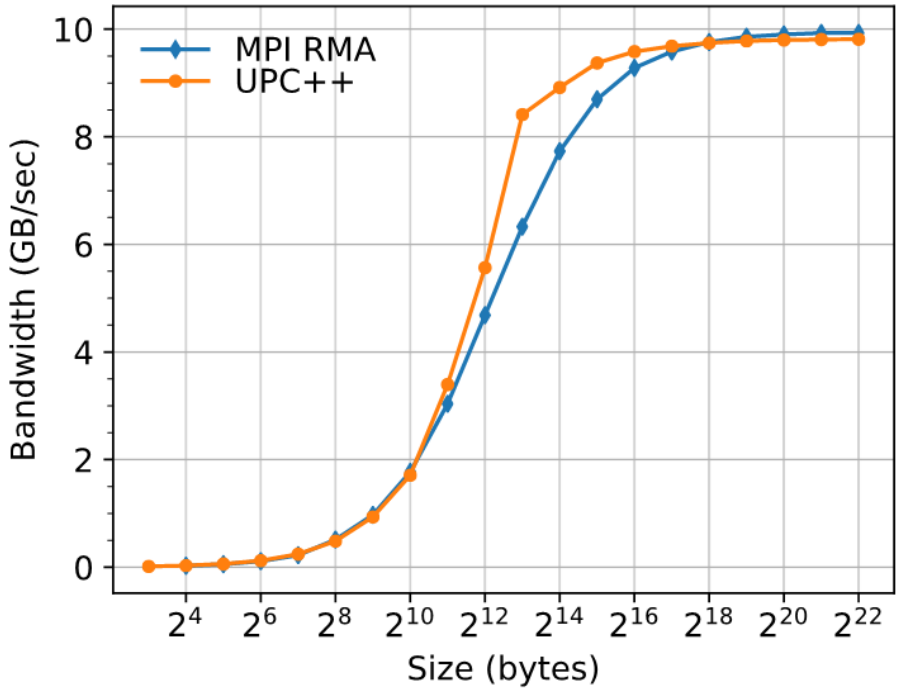


UPC++ on top of GASNet

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)

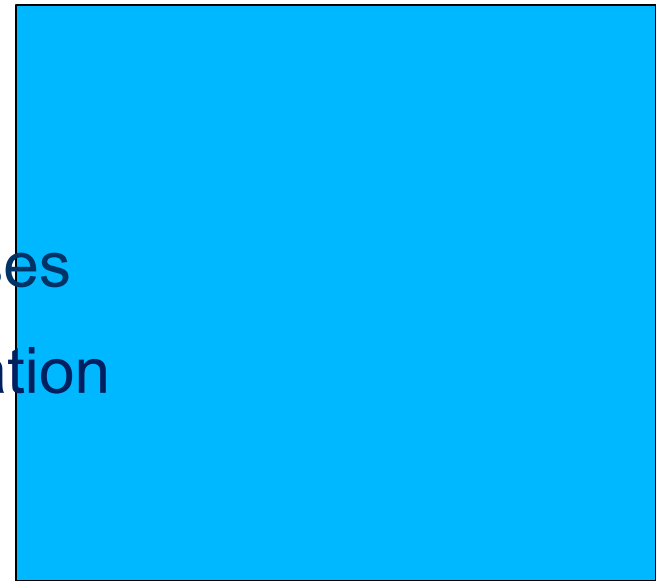
What does UPC++ offer?

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



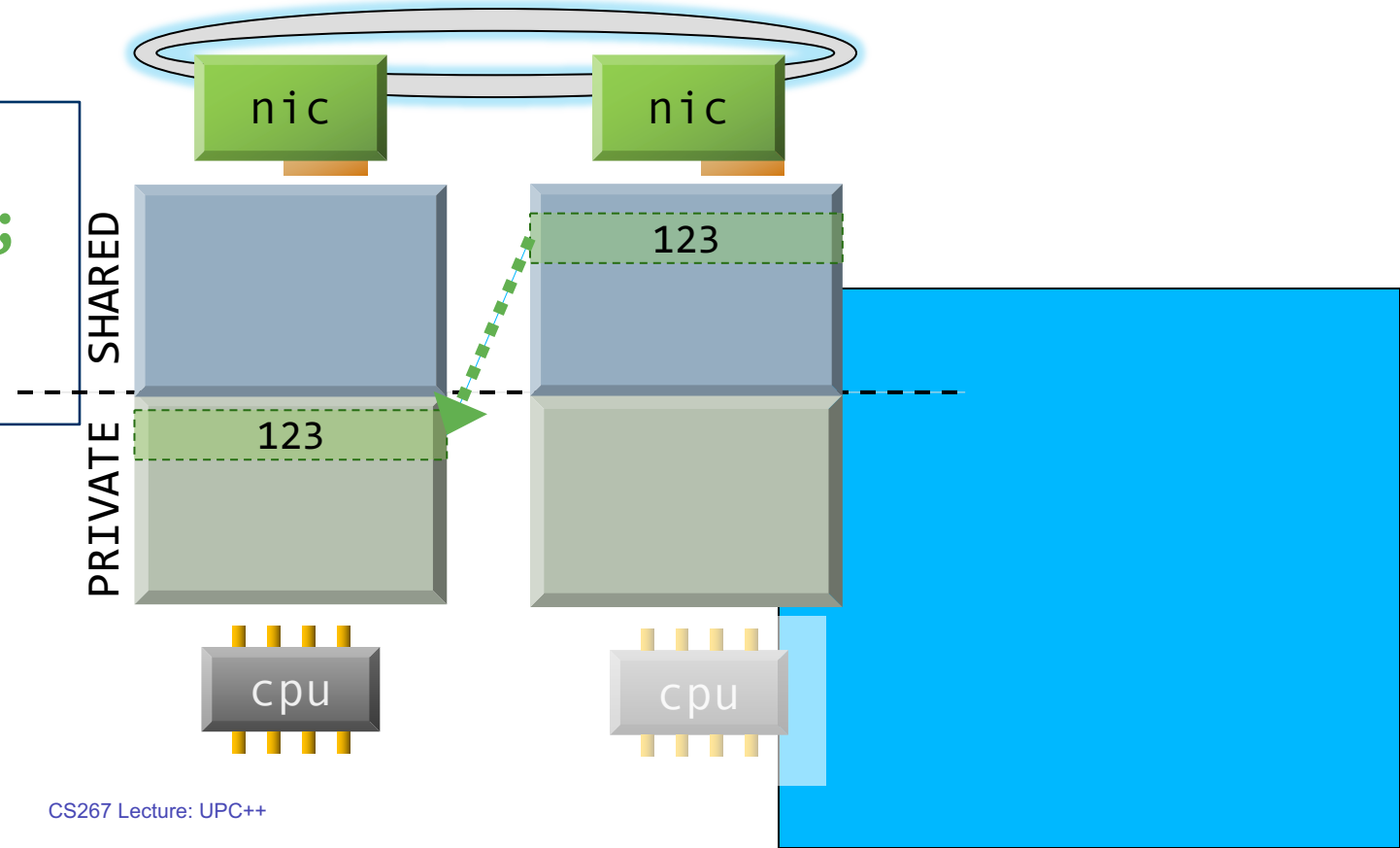
Asynchronous communication (RMA)

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



Example: Monte Carlo Pi Calculation

Estimate Pi by throwing darts at a unit square

Calculate percentage that fall in the unit circle

Area of square = $r^2 = 1$

Area of circle quadrant = $¼ * \pi r^2 = \pi/4$

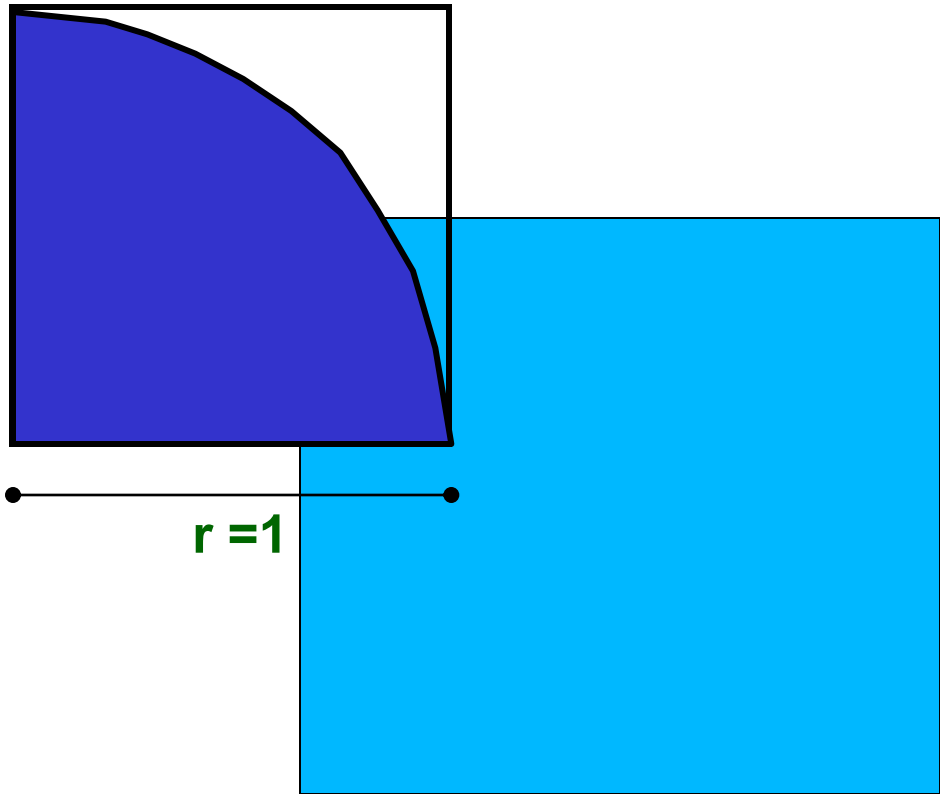
Randomly throw darts at x,y positions

If $x^2 + y^2 < 1$, then point is inside circle

Compute ratio:

points inside / # points total

$\pi = 4*\text{ratio}$



Pi in UPC++

Independent estimates of pi:

```
int main(int argc, char **argv) {
  upcxx::init();
  int hits, trials = 0;
  double pi;

  if (argc != 2) trials = 1000000;
  else trials = atoi(argv[1]);

  generator.seed(upcxx::rank_me()*17);

  for (int i=0; i < trials; i++) hits += hit();
  pi = 4.0*hits/trials;
  cout << "PI estimated to " << pi << endl;
  upcxx::finalize();
}
```

Each rank gets its own copy of these variables

Each rank can use input arguments

Initialize random in math library

Each rank calls "hit" separately

C++ Helper Code for Pi (in C++11)

Required includes and variables:

```
#include <iostream>
#include <random>
#include <upcxx/upcxx.hpp>
default_random_engine generator;
uniform_real_distribution<> dist(0.0, 1.0);
```

Function to throw dart and calculate where it hits:

```
int hit() {
  double x = dist(generator);
  double y = dist(generator);
  if (x*x + y*y <= 1.0) {
      return 1;
  } else {
      return 0;
  }
}
```

UPC++ allows full use of the C++ Standard Template Library

Private vs. Shared Memory in UPC++

Normal C++ variables and objects are allocated in the private memory space for each thread

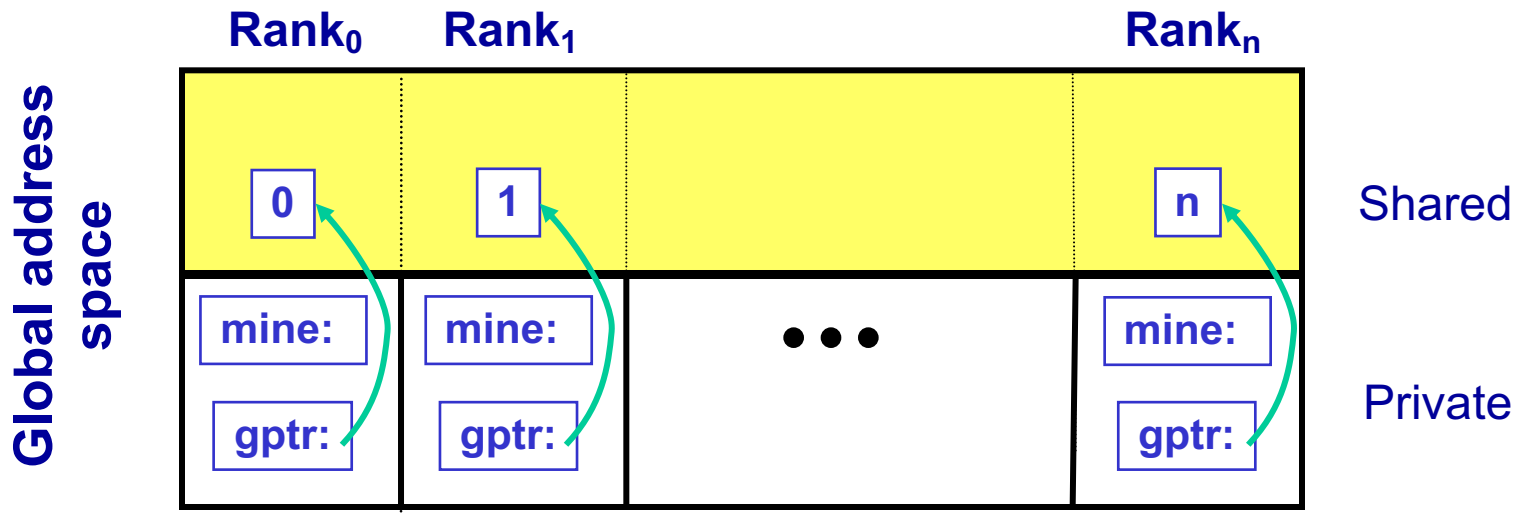
Allocate in shared space with **`new_`**

Free with **`delete_`**

- There are also array versions

upcxx:: qualifier elided from here on out
UPC++ names in green

```
int mine;
global_ptr<int> gptr = new_<int>(rank_me());
```



Broadcast in UPC++

To write an interesting program, we need to have global pointers refer to remote data

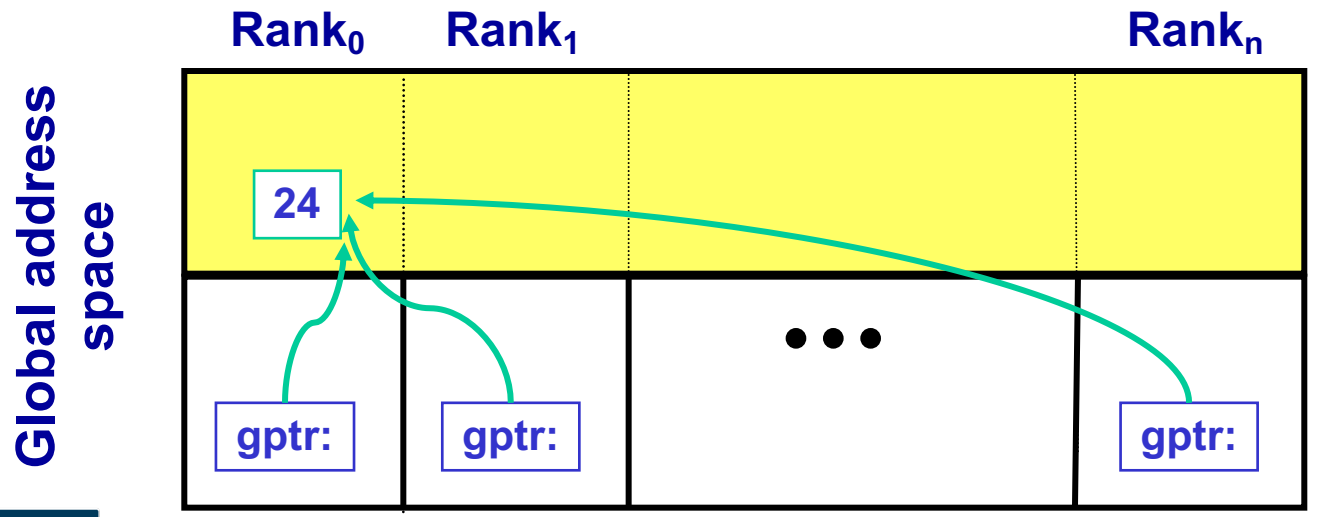
One approach is to broadcast the pointer

```
global_ptr<int> gptr =
      broadcast(new_<int>(24),0).wait();
```

New int variable in shared space, set to 24

broadcast from rank 0

Will explain wait…



Remote access: Put / Get

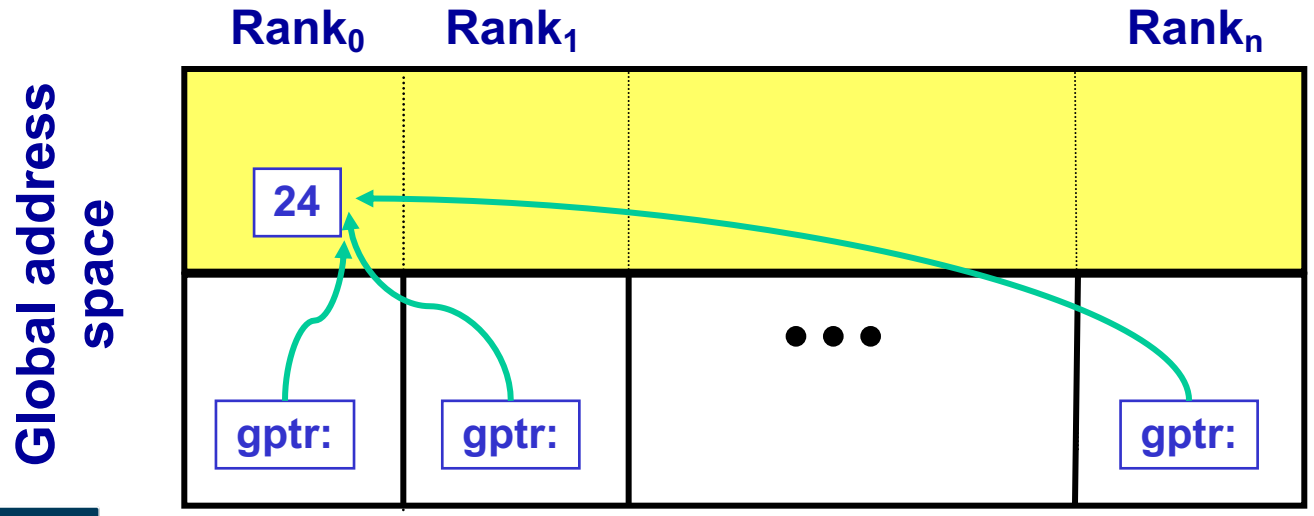
We access shared memory from a remote rank

- Remote get – read a variable on a remote rank
- Remote put – write a variable on a remote rank

Both done via a global pointer to the remote variable

But will take microseconds at best!

- And for most of that time, the processor is just waiting...



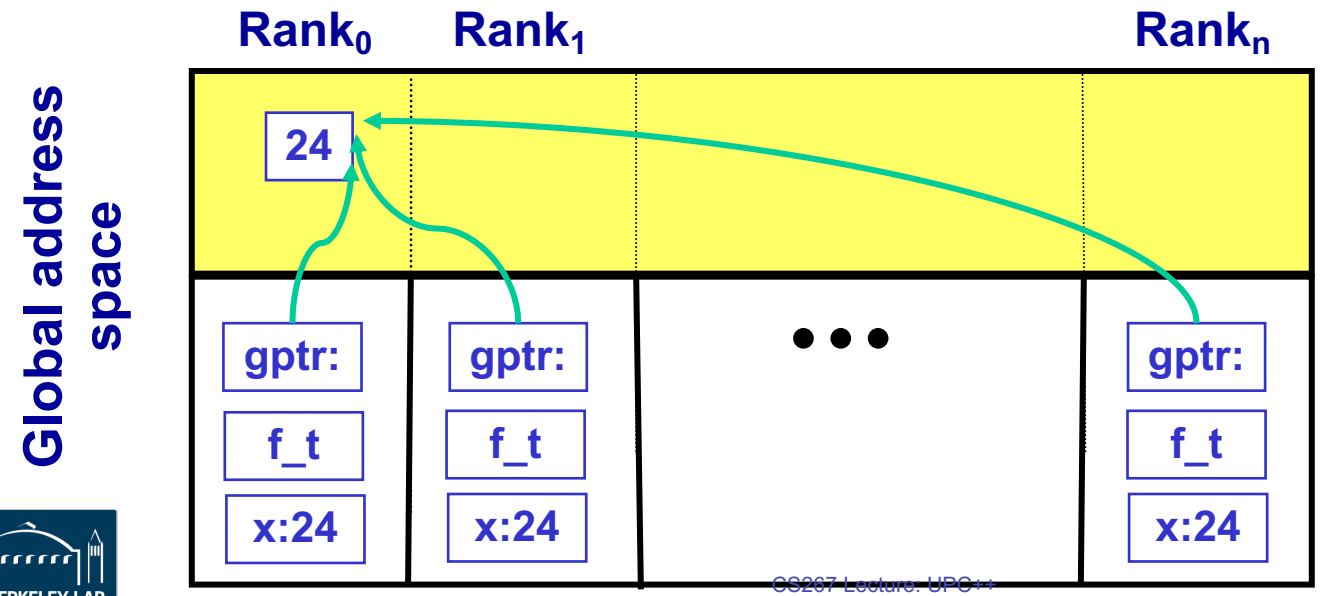
Asynchronous remote operations

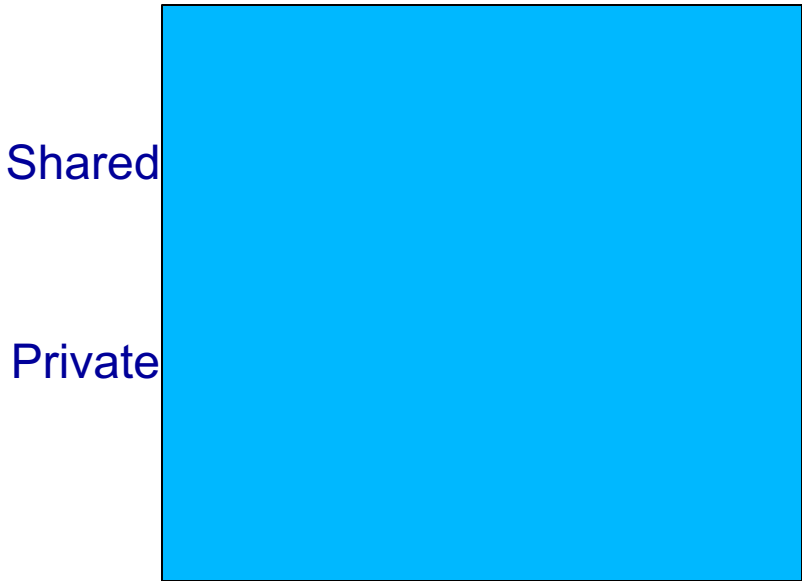
Asynchronous execution used to hide remote latency

Asynchronous get: start reading, but how to tell if you're done?

Put the results into a special "box" called a future

```
future<int> fut_temp = rget(gptr);
// ... Do something expensive
int x = fut_temp.wait();
// all ranks have x = 24
```





Futures in general

A *future* holds a sequence of values and a state (ready / not ready)
Waiting on the returned future lets user tailor degree of asynchrony

```
future<T> f1 = rget(gptr1);   // asynchronous op
future<T> f2 = rget(gptr2);
bool ready = f1.ready();      // non-blocking poll
if !ready …                   // unrelated work...
T t = f1.wait();              // waits if not ready
```

UPC++ has no *implicit* blocking

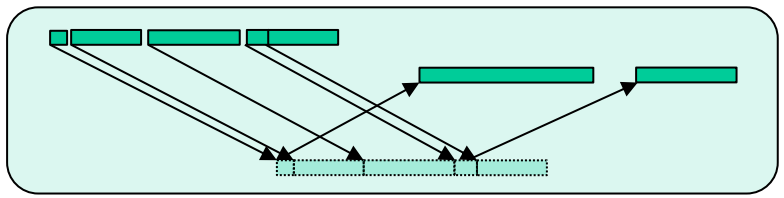
Except for synchronizing operations like “wait” others are implicitly nonblocking (asynchronous)

Large (vector) or non-contiguous Put/Get

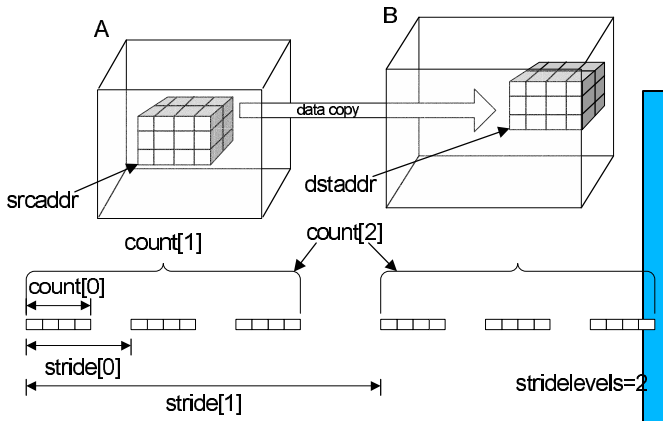
- Vector put and get:

```
int *local = /* ... */;
future<> fut3 = rget(remote, local, size);
fut3.wait();
future<> fut4 = rput(local, remote, size);
fut4.wait();
```

- More operations available for strided and indexed put/get



"gather" style



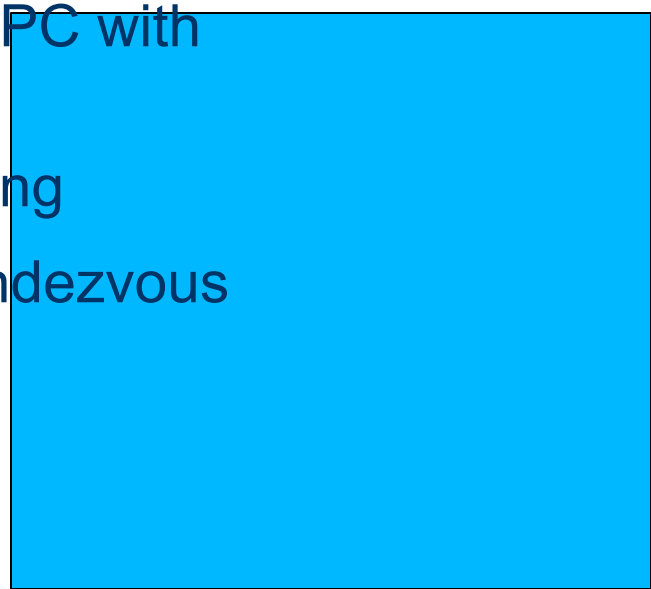
- And for serialization (packing/unpacking) – see RPCs

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)



UPC++ Synchronization

UPC++ has two basic forms of barriers:

1) Synchronous Barrier: block until all other threads arrive (usual)

```
barrier();
```

2) Asynchronous barriers

```
future<> f =
   barrier_async(); // this thread is ready for barrier
// do computation unrelated to barrier
wait(f);      // wait for others to be ready
```

- Reminder: slides elide the upcxx:: that precedes these

Pi in UPC++: Shared Memory Style

Parallel computing of pi, but with a bug

```
int main(int argc, char **argv) {
  init();
  int trials = atoi(argv[1]);
  int my_trials = (trials+rank_me())/rank_n();
  global_ptr<int> hits =
    broadcast(new_<int>(0), 0)).wait();
  generator.seed(rank_me()*17);
  for (int i=0; i < my_trials; i++) {
    int old_hits = rget(hits).wait();
    rput(old_hits+hit(), hits).wait();
  }
  barrier();
  if (rank_me() == 0)
    cout << "PI estimated to "
         << 4.0*(*hits.local())/trials;
  finalize();
}
```

divide work up evenly

broadcast pointer to shared memory from rank 0

accumulate hits
block on communication

Local (C++) version of global pointer that points locally

What is the problem with this program?

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> grid_pptr;
double *grid;


void make_grid(size_t N) {
  grid_gptr = new_array<double>(N);
  grid = grid_gptr.local();
}
```

Downcasting can be used to optimize for co-located processes that share physical memory

Atomics in UPC++

Atomics are indivisible read-modify-write operations

As if you put a lock around each operation, but may have hardware support (e.g., within the network interface)

Create “atomic domain” for shared ints that works with fetch-and-add and load operations.

```
atomic_domain<int> ad_int({atomic_op::load,
                           atomic_op::fetch_add});
int my_hits = 0;
for (int i=0; i < my_trials; i++) my_hits += hit();
ad_int.fetch_add(hits, my_hits,
memory_order_relaxed).wait();
```

Locally “throw darts”

Atomically update shared hits variable

C++ memory order: relaxed says other memory accessed not constrained

```
barrier();
if (rank_me() == 0)
  cout << "PI estimated to " << 4.0*ad_int.load(
          hits, memory_order_relaxed).wait()
          /trials;
```

Once a variable is used with atomics, always use atomics

UPC++ Collectives

UPC++ has a small set of collectives (so far) that are all asynchronous

Broadcast (value, sender): get a value from one rank

```
template <typename T> future <T>
     broadcast (T && value , intrank_t root);
```

Broadcast (buffer, count, sender): get a buffer of values from one rank

```
template <typename T> future <T>
     broadcast (T * buffer, std::size_t count, intrank_t sender);
```

Reduce (value, op): combine values across ranks

```
template <typename T, typename BinaryOp> future <T>
     reduce_all (T && value, BinaryOp &&op);
```

All of these take an optional “team” of ranks and have more general completion semantics (details in UPC++ guide)

Pi in UPC++: Data Parallel Style w/ Collectives

The previous version of Pi works, but is not scalable:

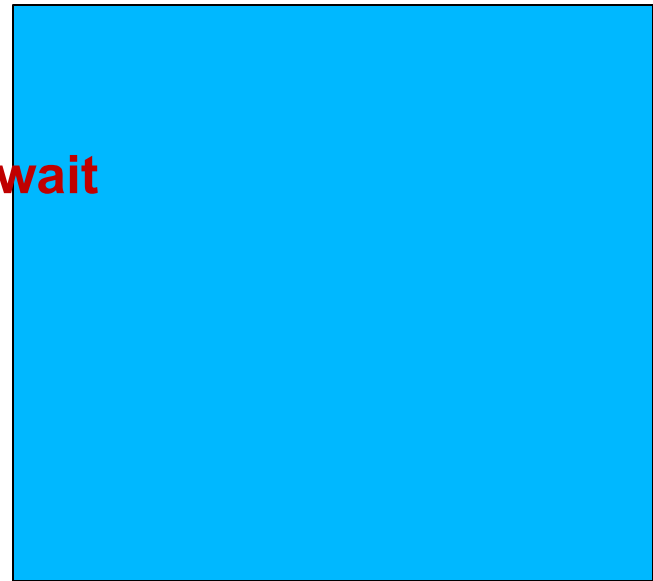
- Updates are serialized on rank 0, ranks block on updates

Use a reduction for better scalability:

```
// int hits;
int main(int argc, char **argv) {
  ...
  for (int i=0; i < my_trials; i++)
     my_hits += hit();
  int hits = reduce_all(my_hits,
                        op_fast_add).wait();
  // barrier();
  if (rank_me() == 0)
    cout << "PI: " << 4.0*hits/trials;
  finalize();
}
```

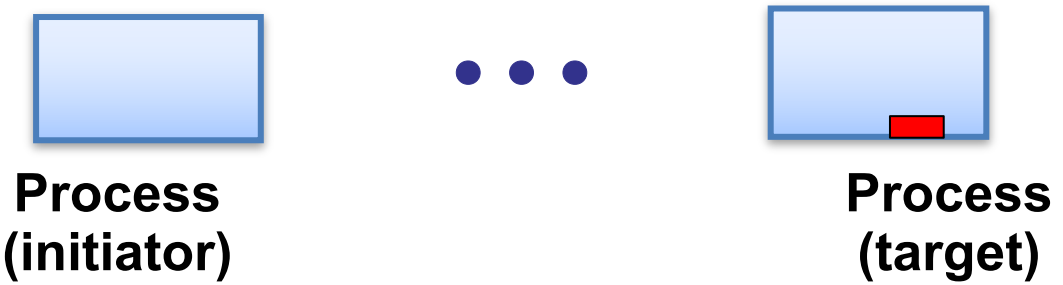
no global variables or shared memory

barrier implied by reduce +wait



Remote procedure call (RPC)

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



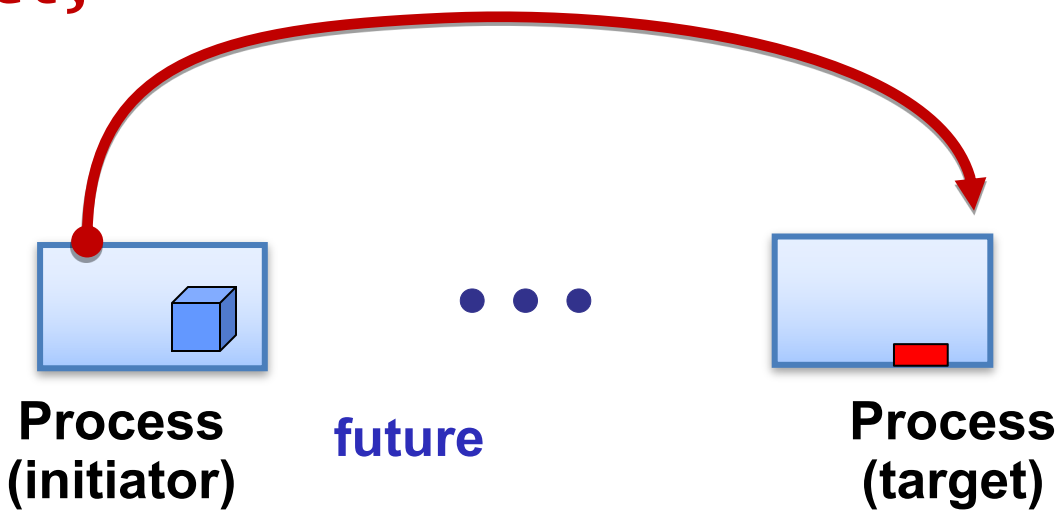
Remote procedure call (RPC)

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

1. Initiator injects the RPC to the *target* process

1

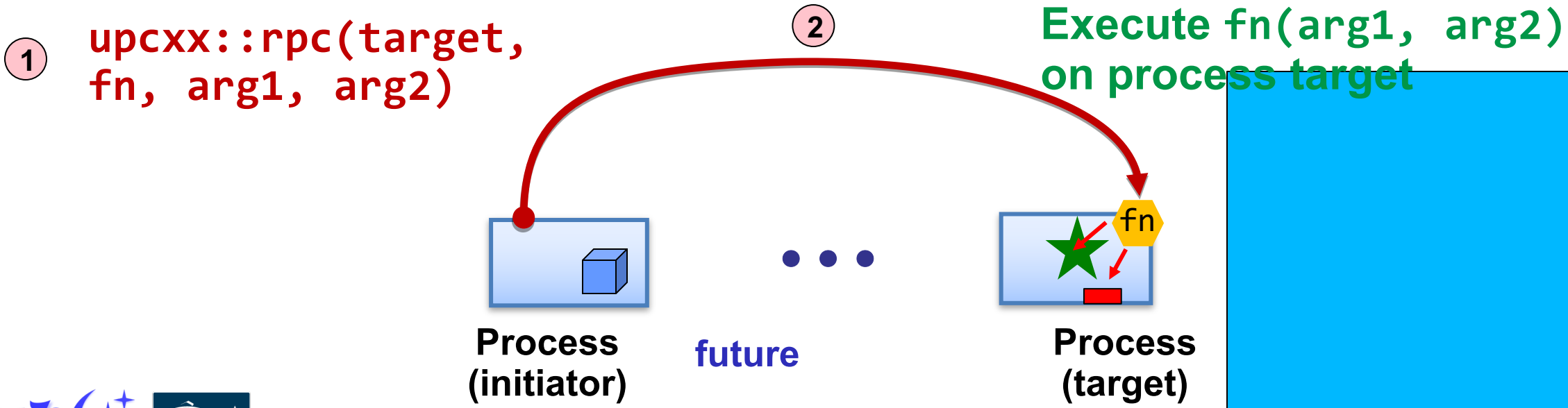
```
upcxx::rpc(target,
fn, arg1, arg2)
```



Remote procedure call (RPC)

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

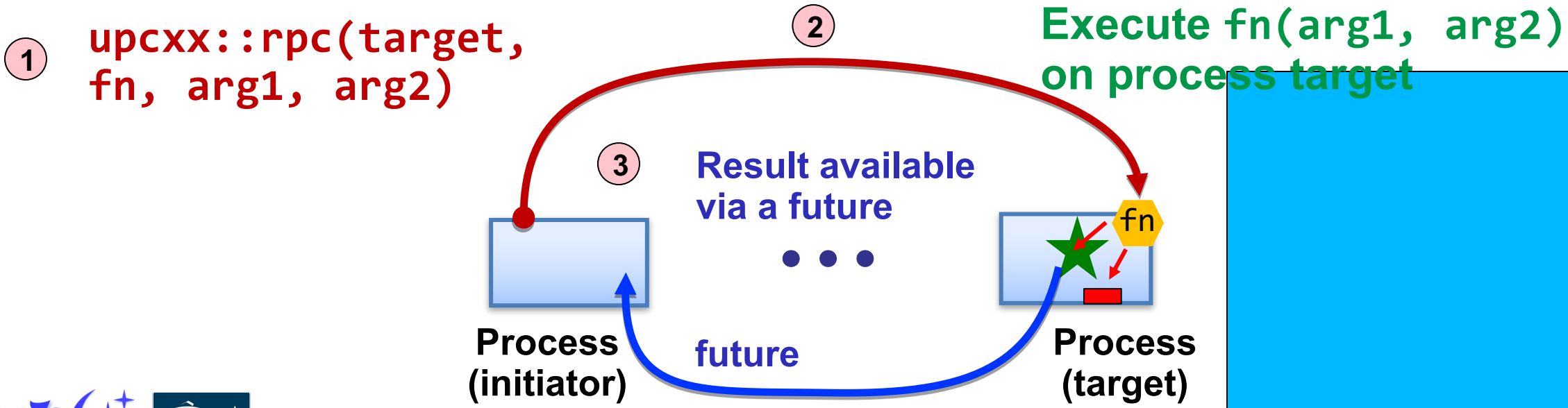
1. Initiator injects the RPC to the *target* process
2. Target process executes `fn(arg1, arg2)` at some later time determined at the target



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

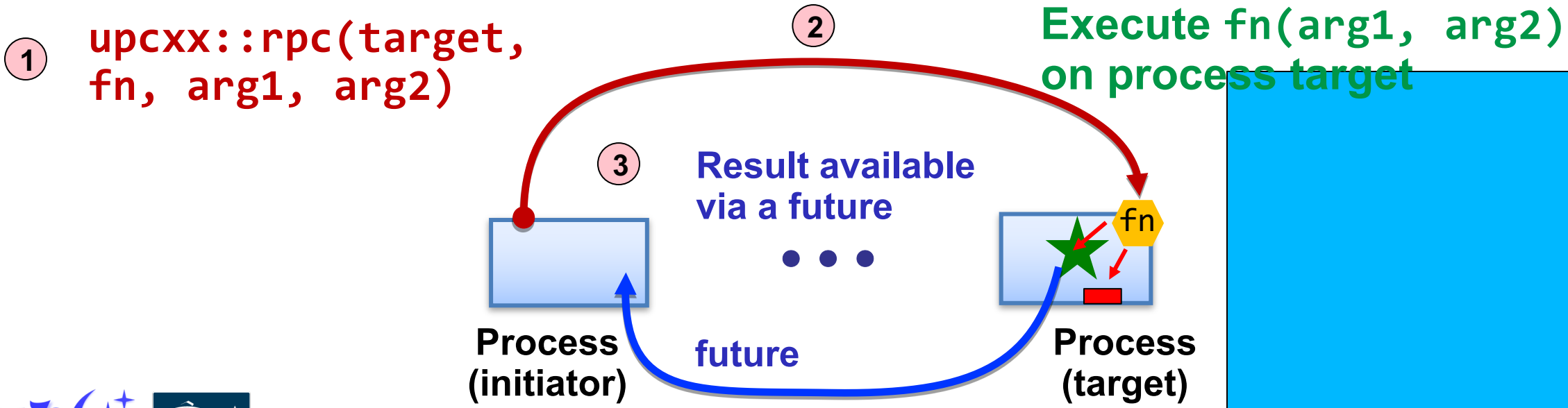


Remote procedure call (RPC)

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



Pi in UPC++: RPC

RPC used to synchronize updates

```
int hits = 0;
int main(int argc, char **argv) {
  init();
  int trials = atoi(argv[1]);
  int my_trials = (trials+rank_me())/rank_n();
  generator.seed(rank_me()*17);
  for (int i=0; i < my_trials; i++) {
    rpc(0, [](int hit) { hits += hit; },
           hit()).wait();
  }
  barrier();
  if (rank_me() == 0)
    cout << "PI estimated to " << 4.0*hits/trials;
  finalize();
}
```

RPC can refer to global variable

send update to rank 0
block on the update

Similar to atomics, but always runs on the remote processor (not in network) and can be arbitrary function (not just simple operation)

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);
  });
```

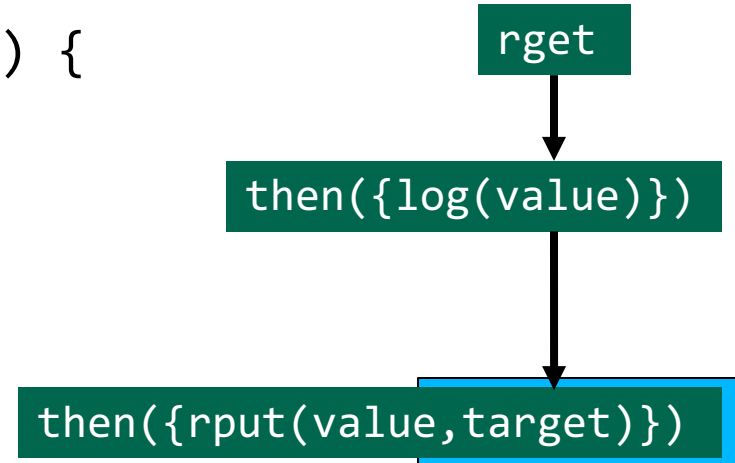
Vector get does not produce a value

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();
```



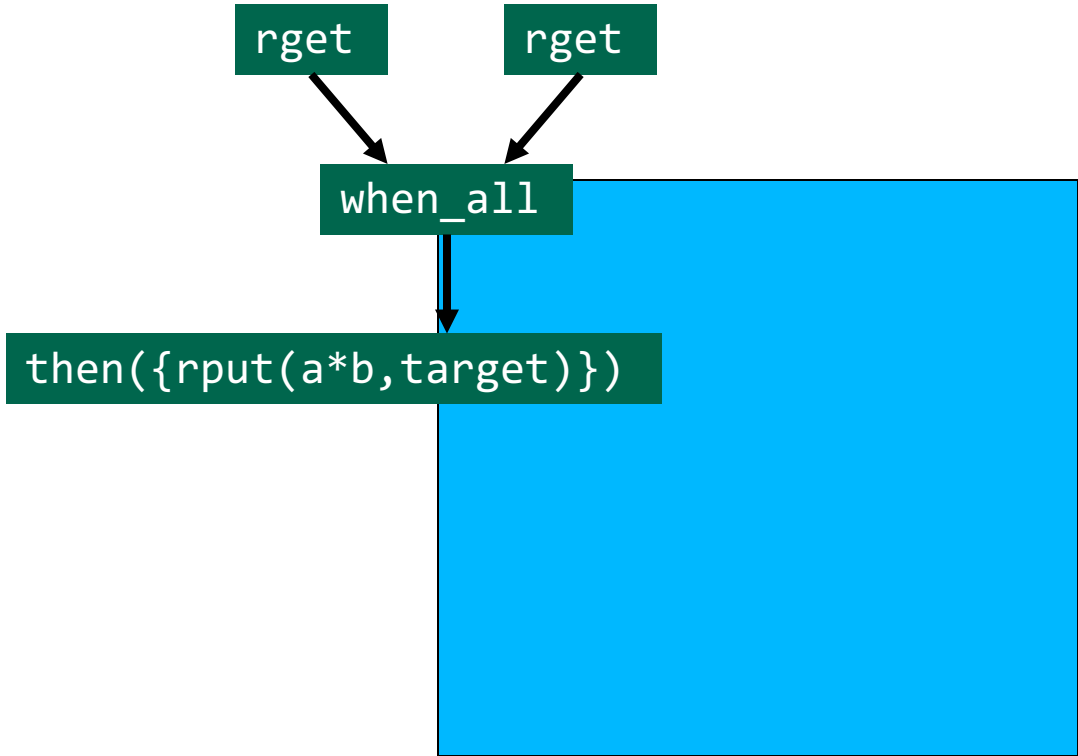
This code retrieves an integer from a remote location, computes its log, and then sends it to a different remote location

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();
```



Distributed objects

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

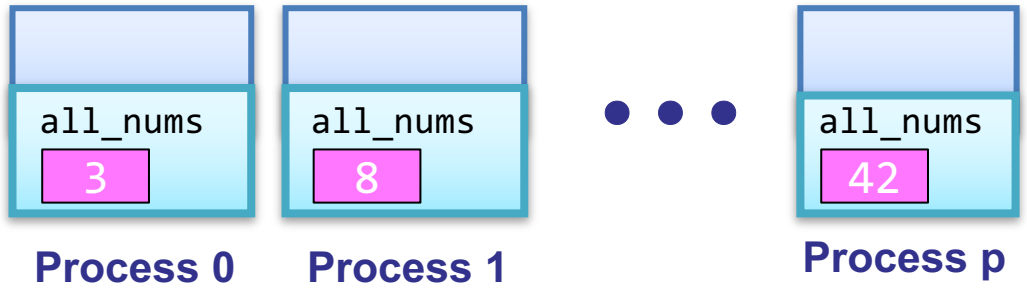
```
dist_object<T>(T value, team &team = world());
```

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

- No communication to set up or tear down
- Scalable metadata representation
- Does not require a symmetric heap
- Can be constructed over teams

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



Pi with a distributed object

A distributed object can be used to store the results from each process

```
// Throws a random dart and returns 1 if it is
// in the unit circle, 0 otherwise.
int hit();

...

dist_object<int> all_hits(0);
for (int i = 0; i < my_trials; ++i)
  *all_hits += hit();
barrier();
if (rank_me() == 0) {
  for (int i = 0; i < rank_n(); ++i)
    total += all_hits.fetch(i).wait();
  cout << "PI estimated to " << 4.0*total/trials;
}
```

Results for each process

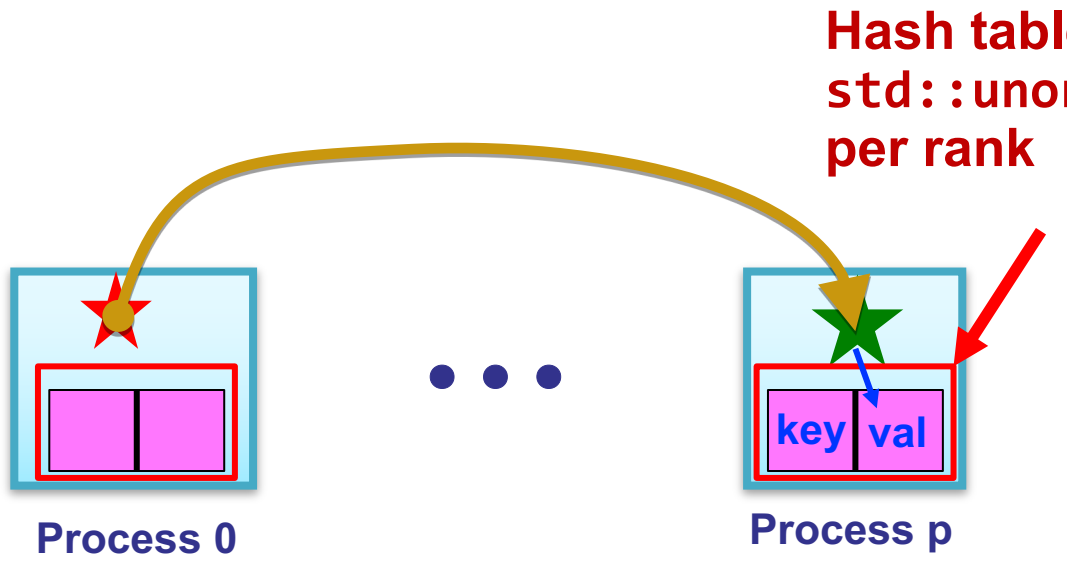
Dereference to obtain this process's value

Obtain another process's value

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

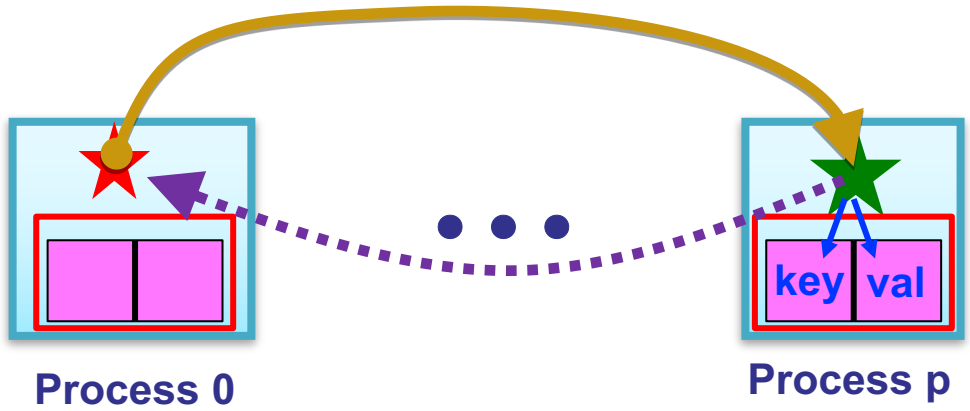
Insertion initiates an RPC to the owner and returns a future that represents completion of the insert

```
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);
}
```

Send RPC to the rank determined by key hash

Key and value passed as arguments to the remote function

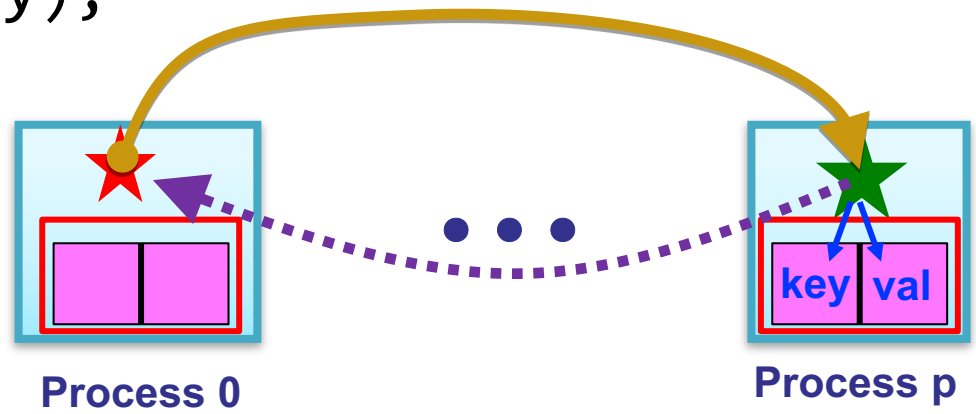
UPC++ uses the distributed object's universal name to look it up on the remote process



DHT find

Find also uses RPC and returns a future

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



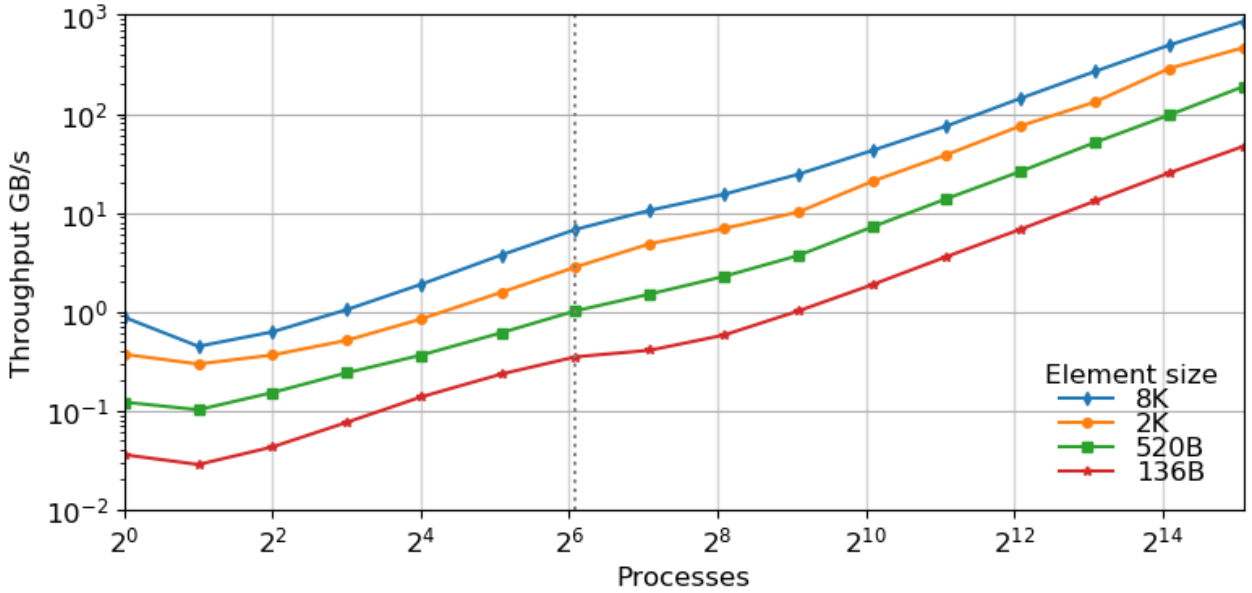
Optimized DHT scales well

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

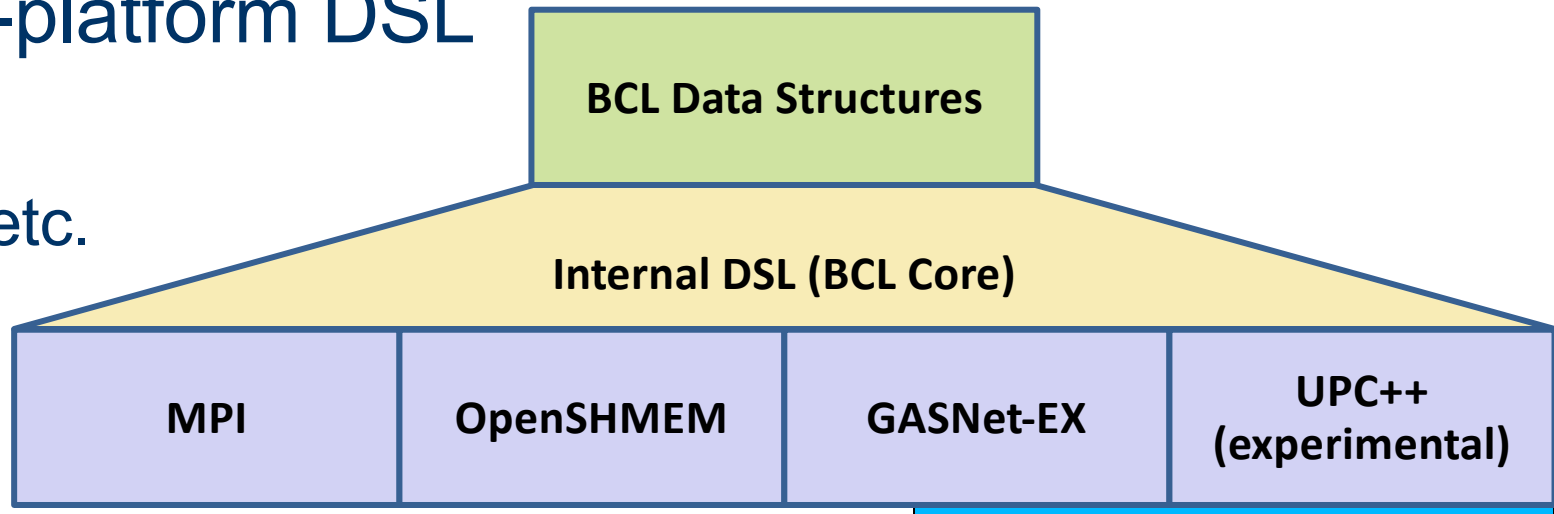


Cori @ NERSC (KNL)

Cray XC40

Berkeley Container Library (Distributed PGAS Data Structures)

- Built on BCL Core, cross-platform DSL
- Data structures for:
  - Hash tables, Bloom filters, etc.
  - Queues
  - Dense and sparse matrices
- Multiple backends
- Adding support for GPUs, more data structures, etc.



Final project opportunities!

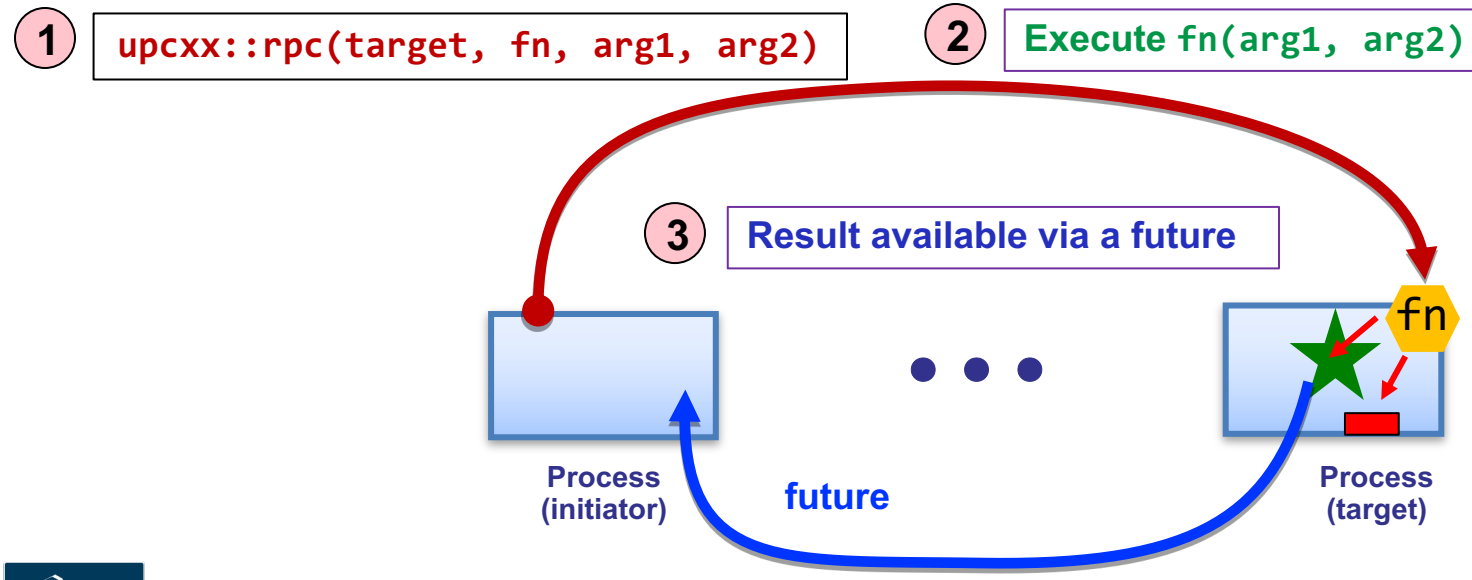
Ben Brock PhD thesis on BCL: github.com › berkeley-container-library

RPC and progress

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

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

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



Progress

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

This design decision keeps the runtime lightweight and simplifies synchronization

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

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

Two levels of progress

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

Invoking user-level progress

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

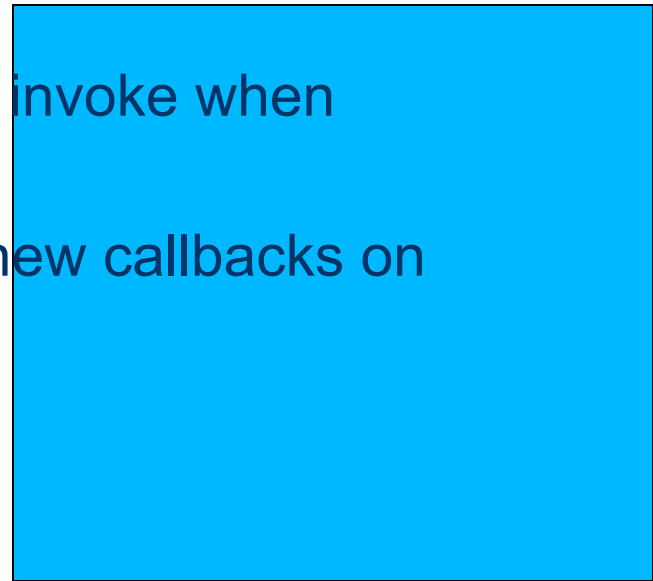
- So do blocking calls such as `wait()` and `barrier()`

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

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

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

- "Some number" could be zero, so may need to periodically invoke when expecting callbacks
- Callbacks may not wait on communication, but may chain new callbacks on completion of communication



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

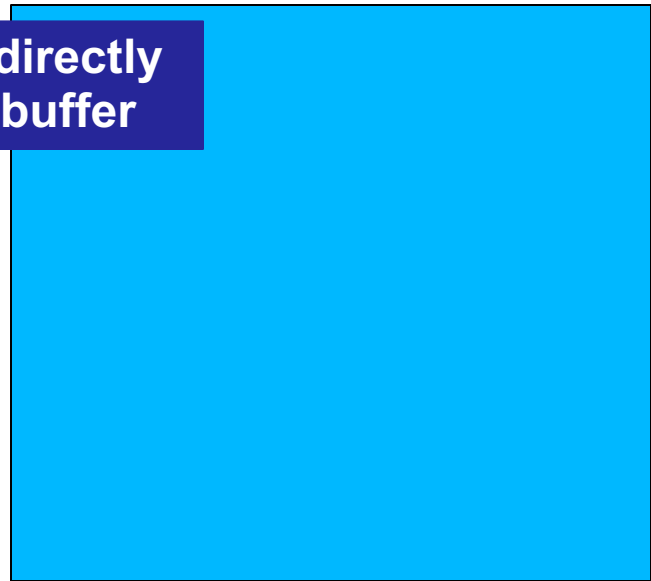
- `view<T>`: non-owning sequence of elements

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

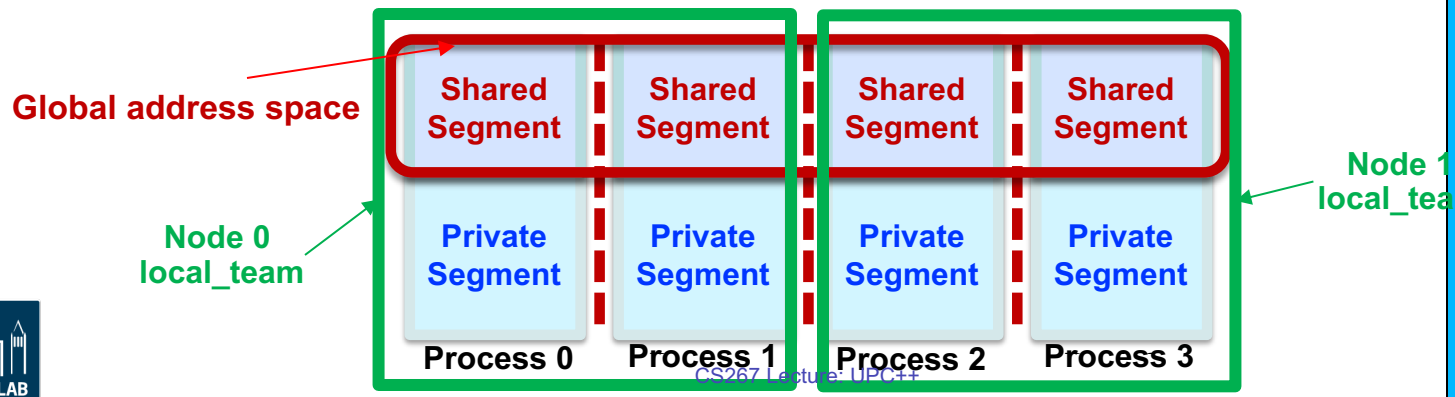
- Some process pairs are “closer” than others
- Ex: cabinet > switch > node > NUMA domain > socket > core

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 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();
```

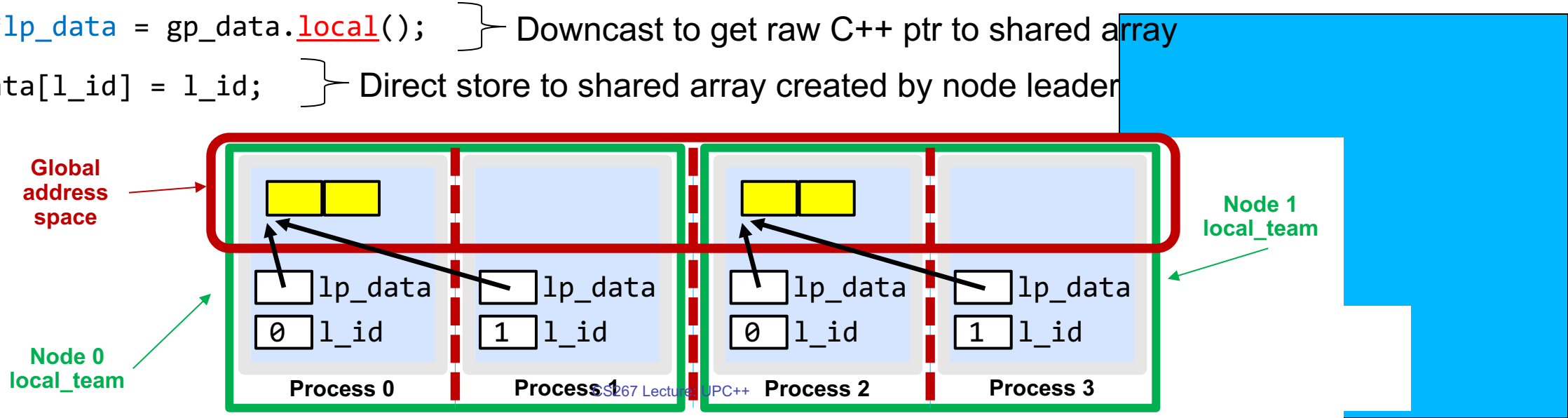
Allocate and share one array **per node**

```
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

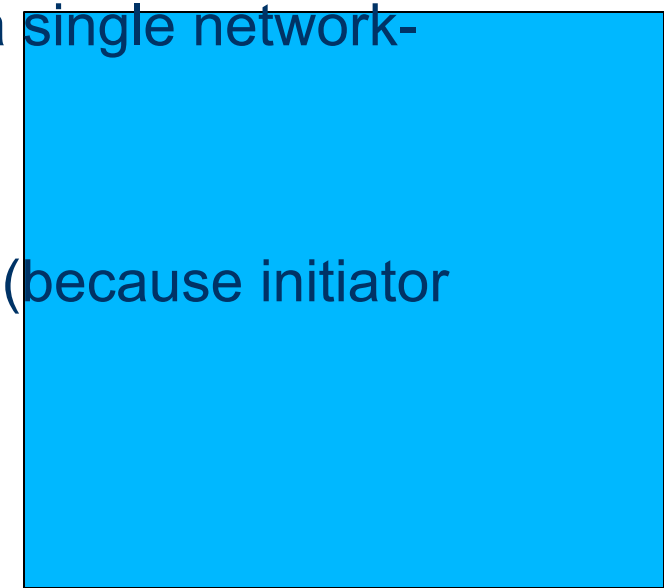
Register an operation on a promise

Completion: "signaling put"

One particularly interesting case of completion:

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

Can point to memory on a local or remote GPU

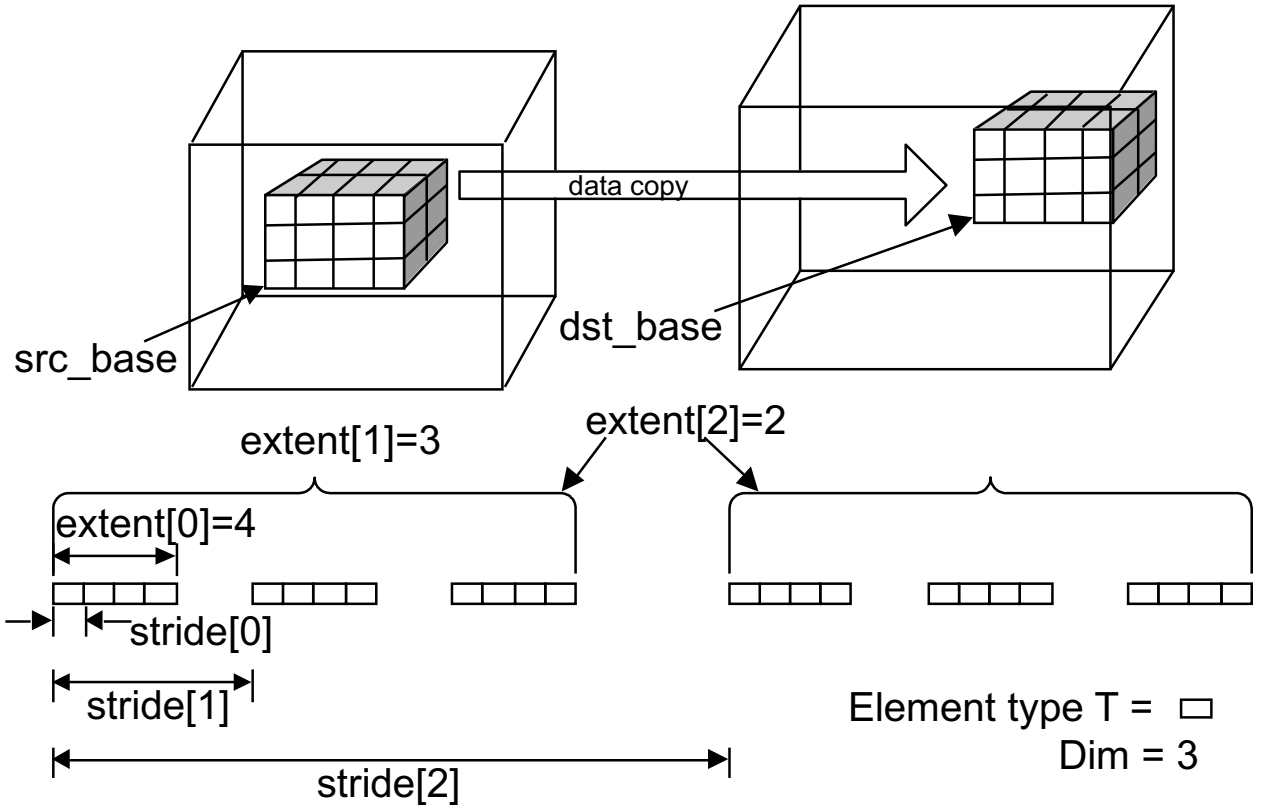
Non-contiguous RMA

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

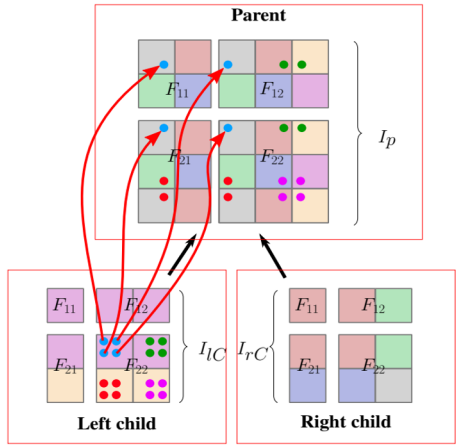
- Irregular: *iovec*-style iterators over pointer+length
- Regular: iterators over pointers with a fixed length
- Strided: N-d dense array copies + transposes

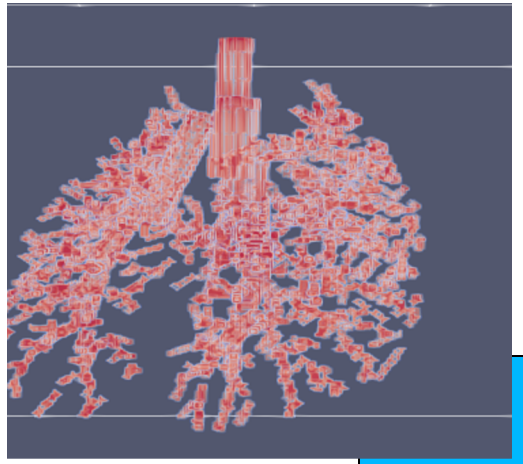
Application Case Studies

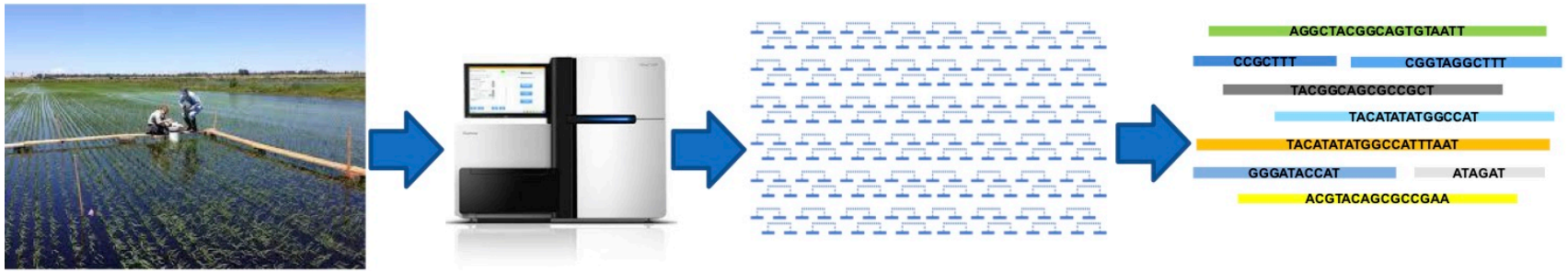
Application case studies

UPC++ has been used successfully in several applications, e.g.:

- symPack, a sparse symmetric matrix solver
- Sim-COV, agent-base simulation of lungs with COVID
- MetaHipMer, a genome assembler







Sparse multifrontal direct linear solver

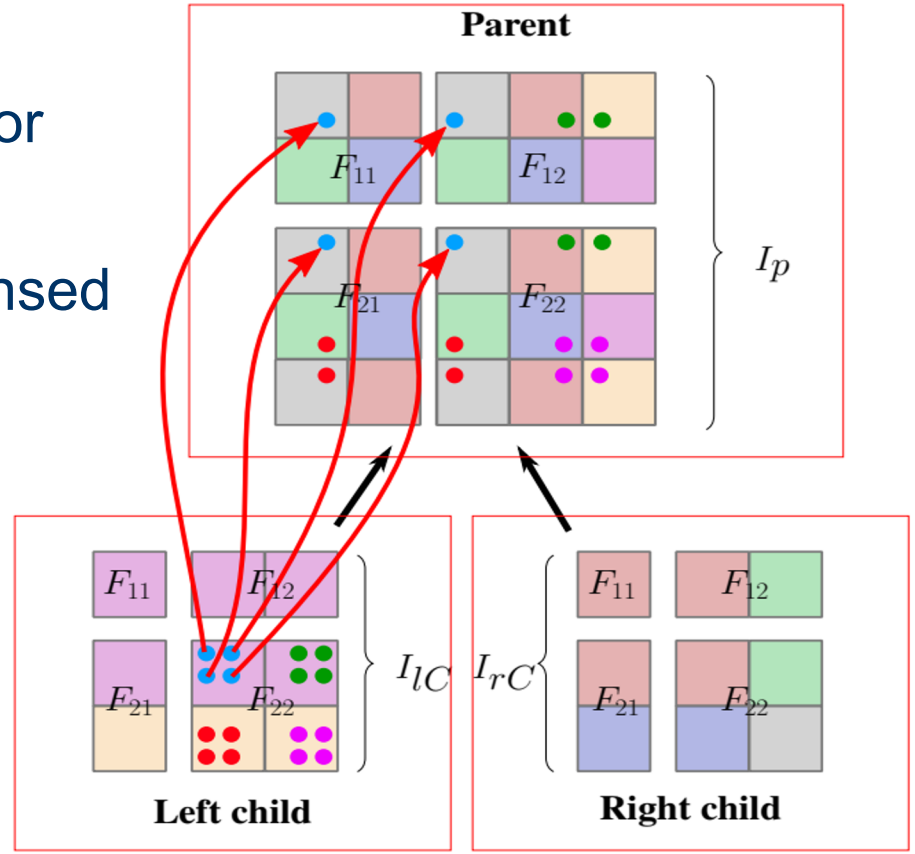
Sparse matrix factorizations have low computational intensity and irregular communication patterns

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**

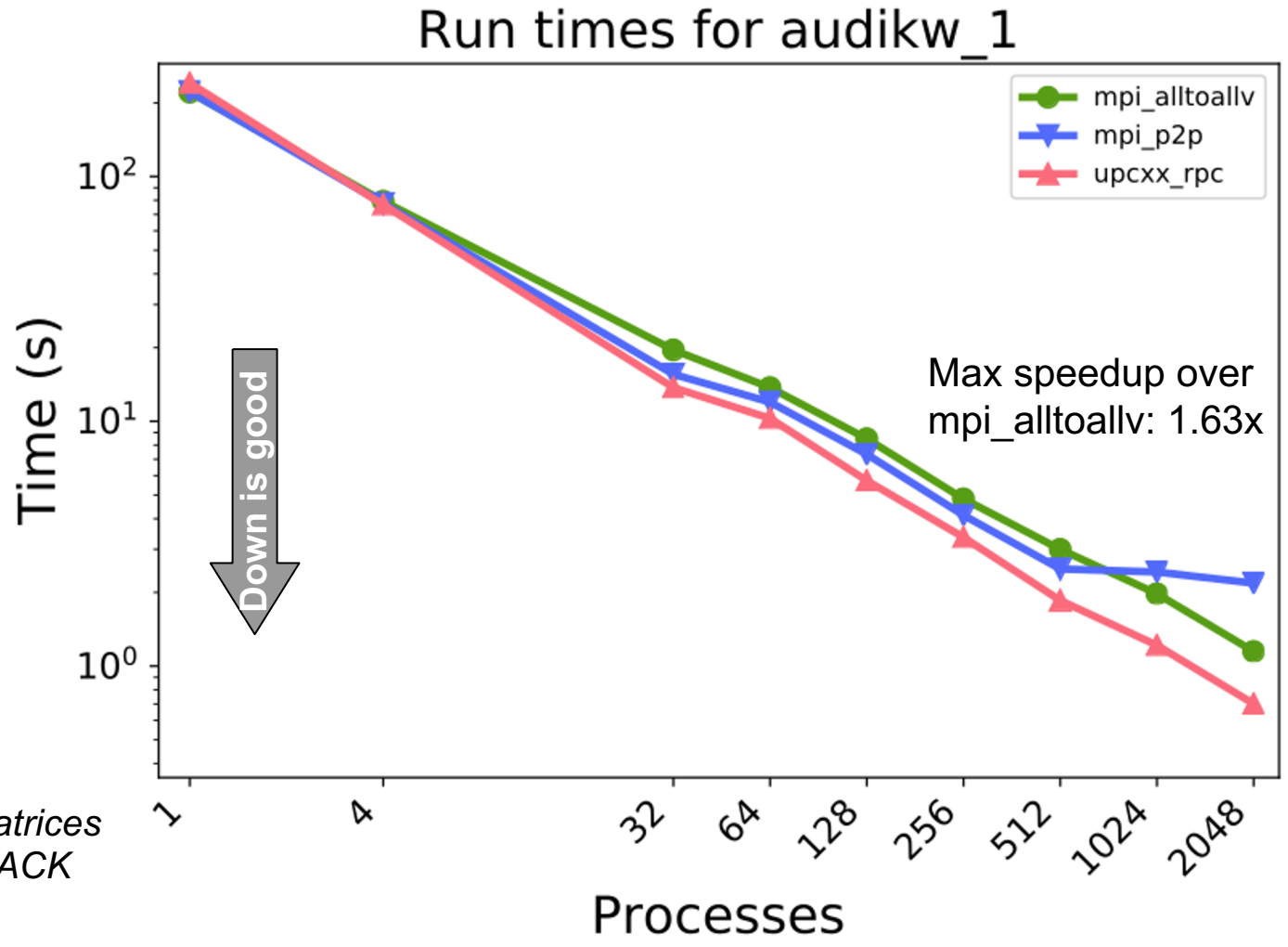
Code available as part of upcxx-extras BitBucket repo



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

UPC++ improves sparse solver performance (extend-add)



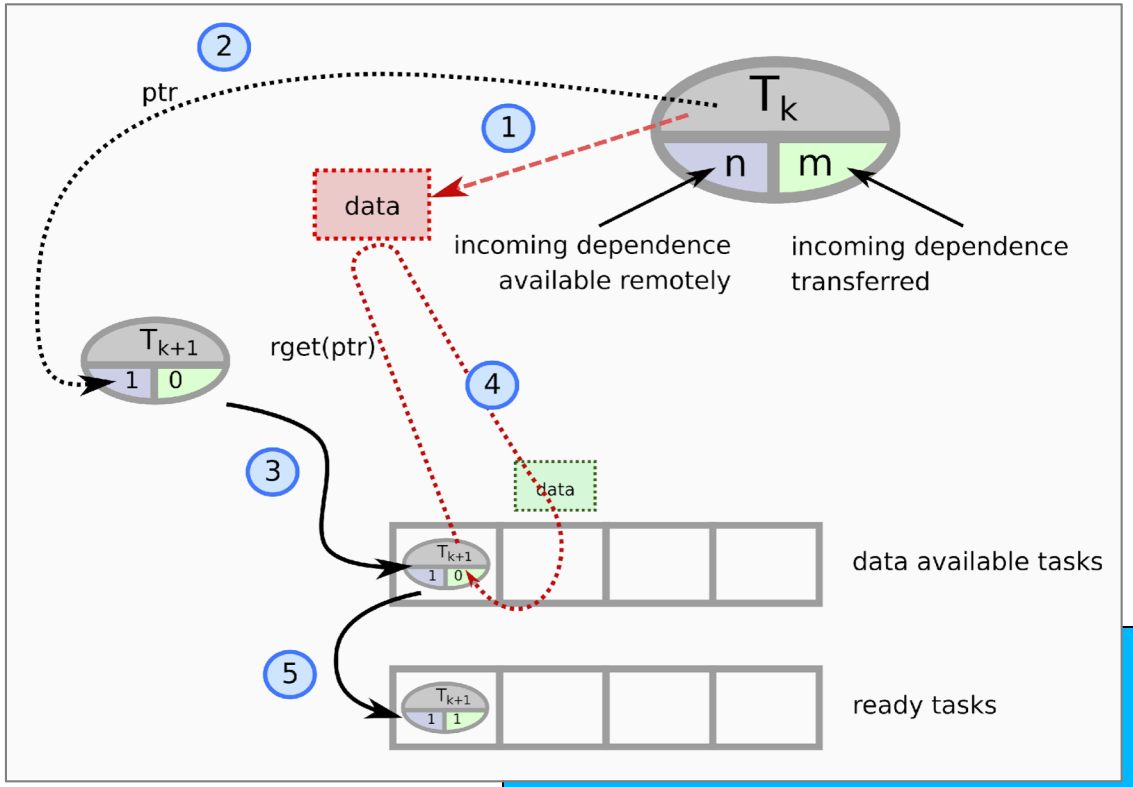
Assembly trees / frontal matrices extracted from STRUMPACK

Experiment done on NERSC Cori KNL Cray XC Aries

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

symPACK: a solver for sparse symmetric matrices

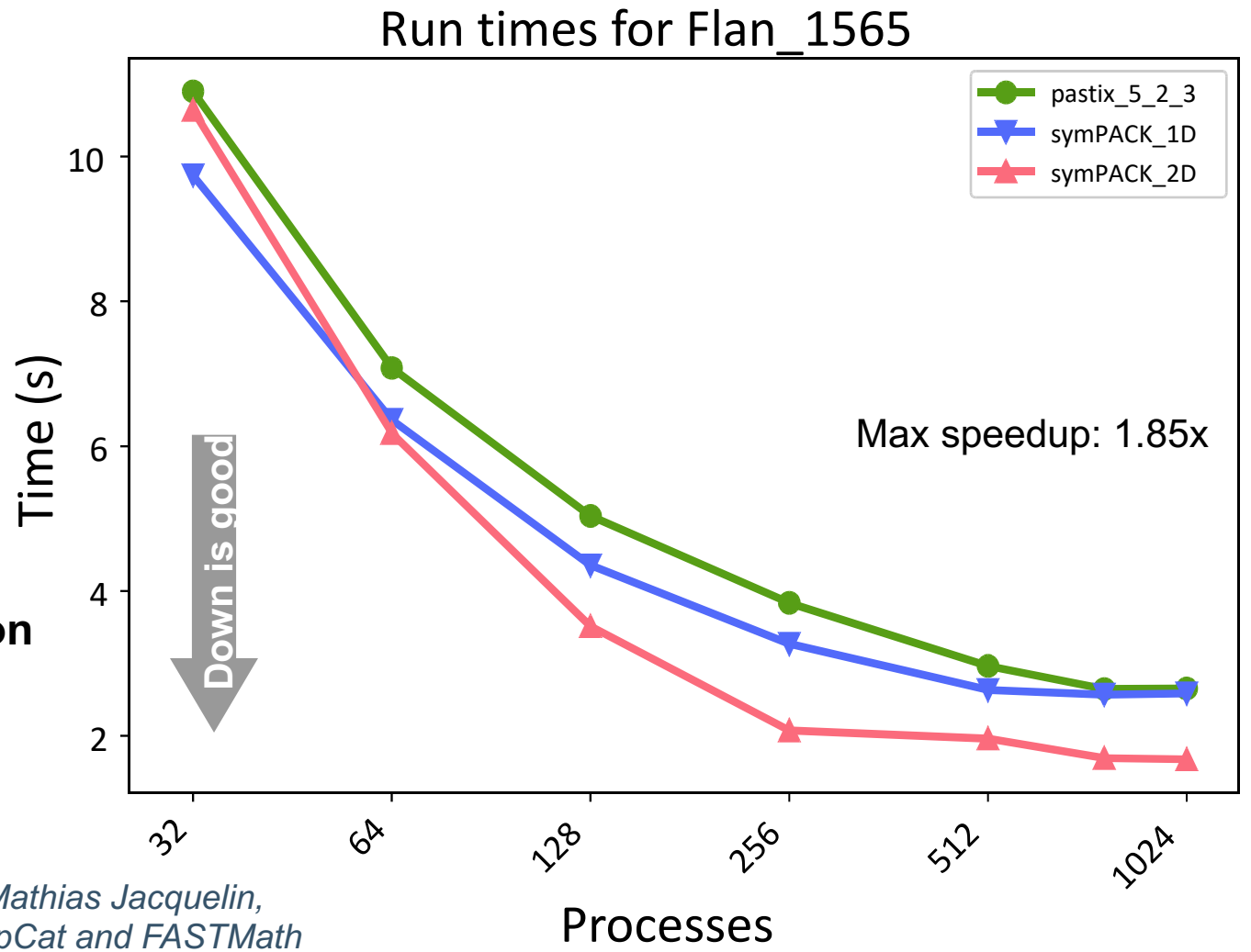
1) Data is produced
2) Notifications using **`upcxx::rpc_ff`**
   - Enqueues a **`upcxx::global_ptr`** to the data
   - Manages dependency count
3) When all data is available, task is moved in the data available task list
4) Data is moved using **`upcxx::rget`**
   - Once transfer is complete, update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



https://upcxx.lbl.gov/sympack

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

symPACK strong scaling experiment



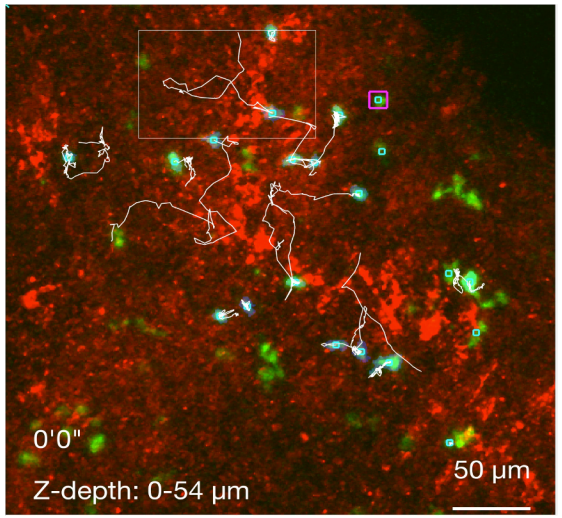
Experiment done on NERSC Cori KNL Cray XC Aries

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

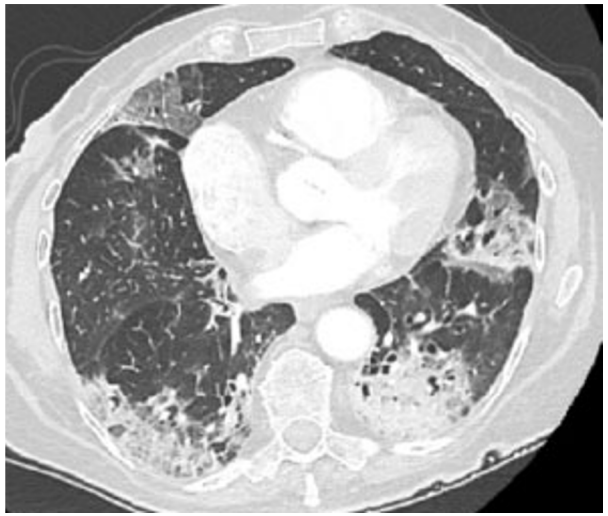
Work and results by Mathias Jacquelin, funded by SciDAC CompCat and FASTMath

SIM-Cov Implementation

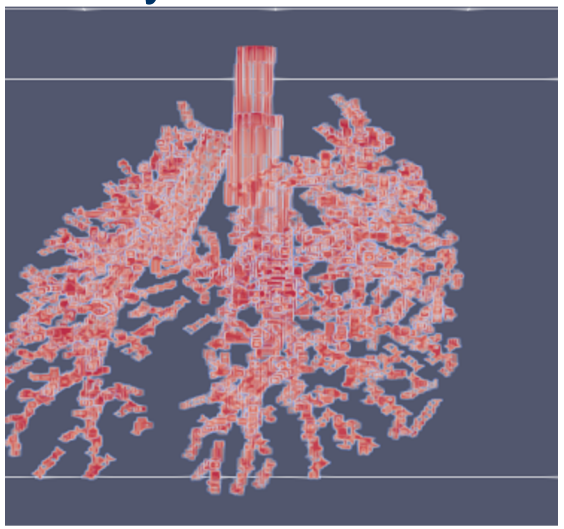
- Goal is to model the entire lung at the cellular level:
  - 100 billion epithelial cells
  - 100s of millions of T cells
  - Complex branching fractal structure
  - Time resolution in seconds for 20 to 30 days
- SIM-Cov in UPC++
  - Particles move over time, but computation is localized
  - Load balancing is tricky: active near infections
  - RPCs for easy of communication overlap and synchronization



Imaging of T cell movement in lung tissue

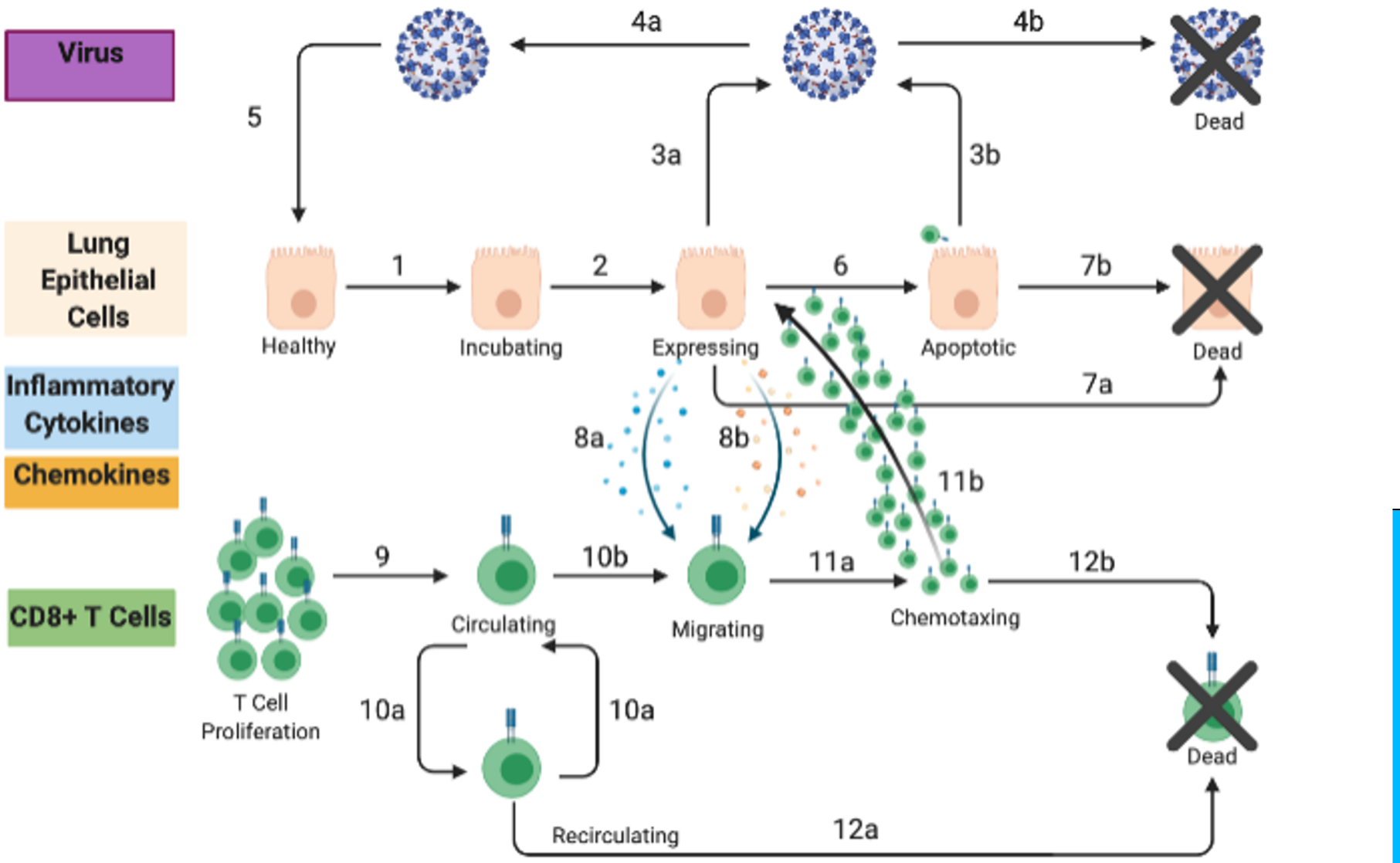


Lung CT showing sites of infection



Fractal model of airways in lung

SIM-Cov Components: 3D Agent-Based Model



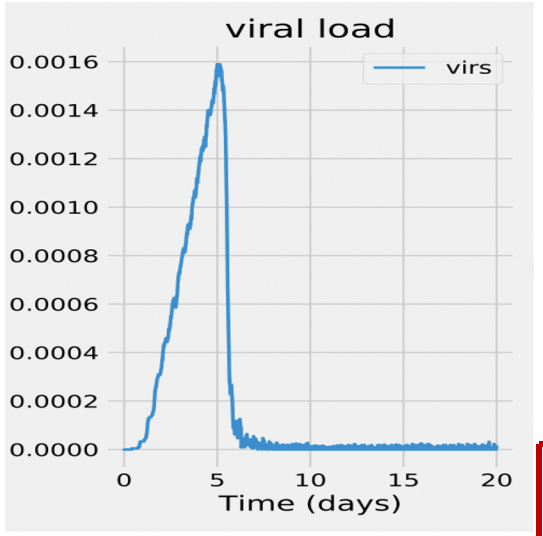
Speculative Simulations to Explore Role of T cells in disease severity

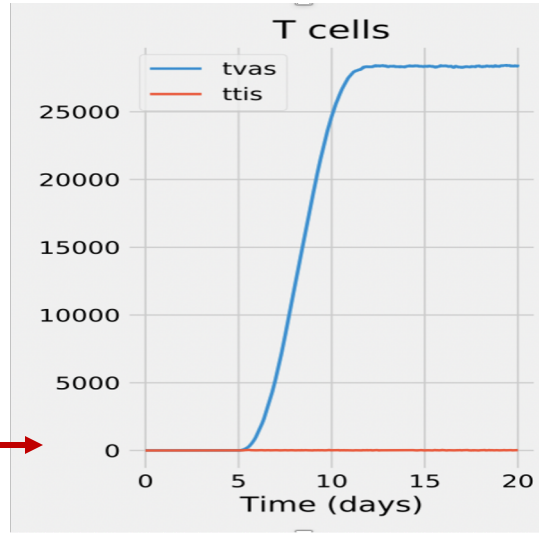
Mild infection:

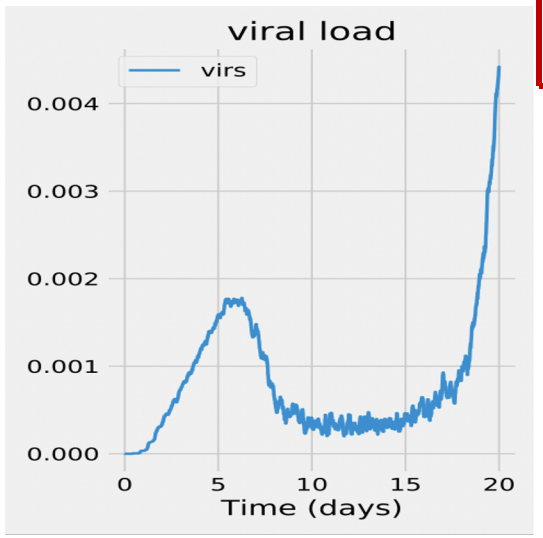
- high T cell response
- controls viral infection
- recovery by day 10 (viral drops near zero)

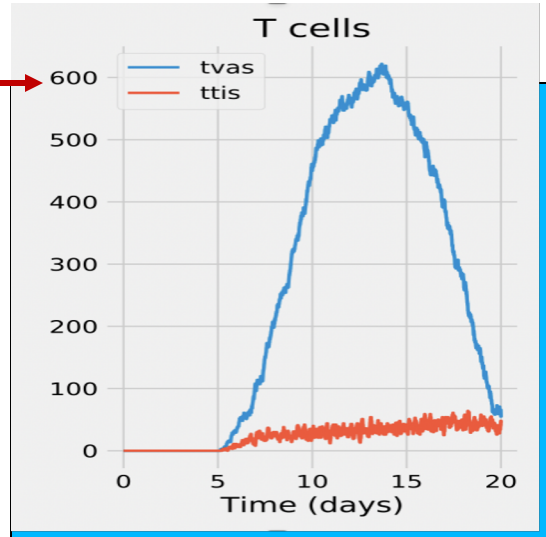
Severe infection:

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



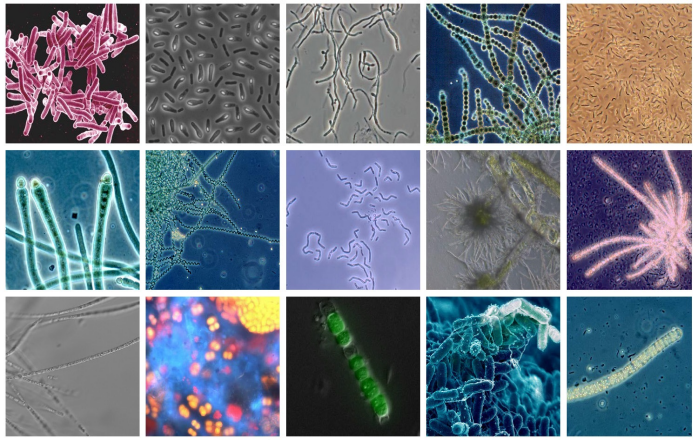






ExaBiome: Exascale Solutions for Microbiome Analysis

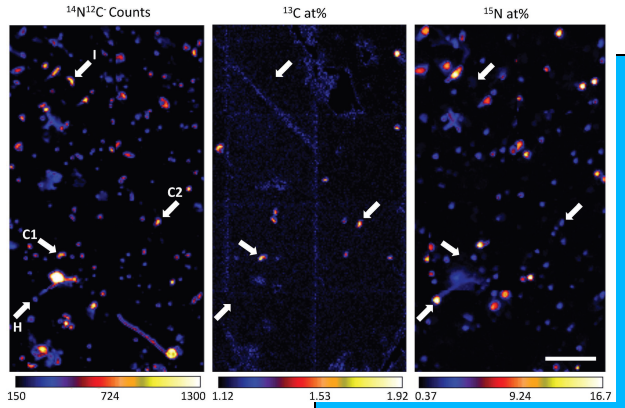
What happens to microbes after a wildfire? (1.5TB)



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

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

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



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

De Novo genome assembly problem

Input

reads
(input, typically 100-250 chars)

GCTACGGAATAAAACCAGGAACAACAGAGCC_AGCAC

ATAAAACCAGGTACAACAGACCCAGCACGGATCCA

GC_ACGGAATACAACCAGGAACAACAGACCCAGCAC

GAACAACAGACCCAGCATGGATCCA

Multiple copies
(20x typical)

errors

GCTACGGAATAAAACCAGGAACAACAGACCCAGCACGGATCCA

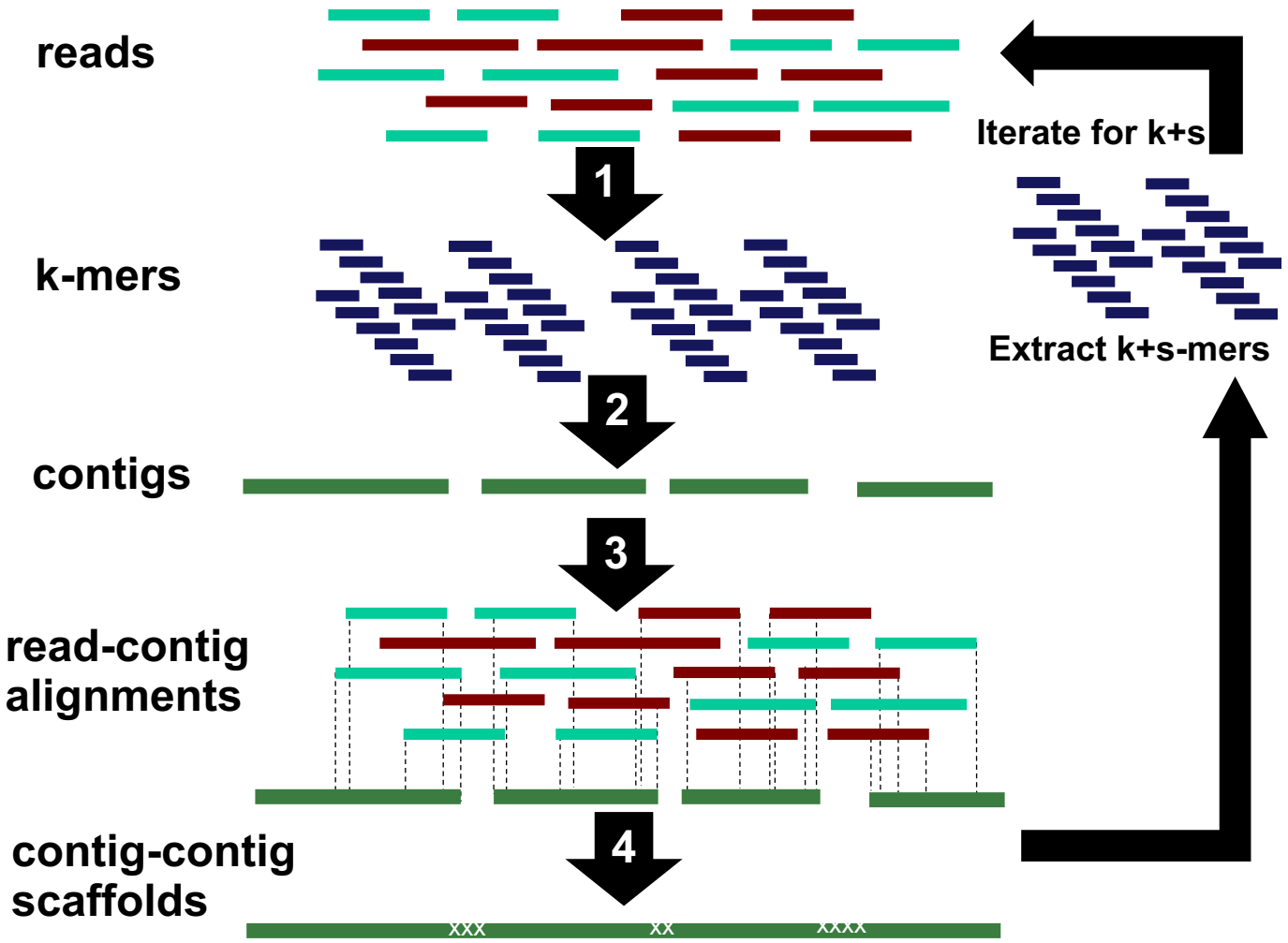
Output

Assembled genome (or 10s of Ks of bp fragments so we can find genes, etc.)

Understanding an environmental microbiome

Best paper finalist at Supercomputing 18

(Meta)HipMer (Meta)Genome Assembly



1) K-mer Analysis

Histogram

2) Contig Generation

Connected components

Distributed Hash Table

3) Alignment

Read/contig alignment

Smith Waterman

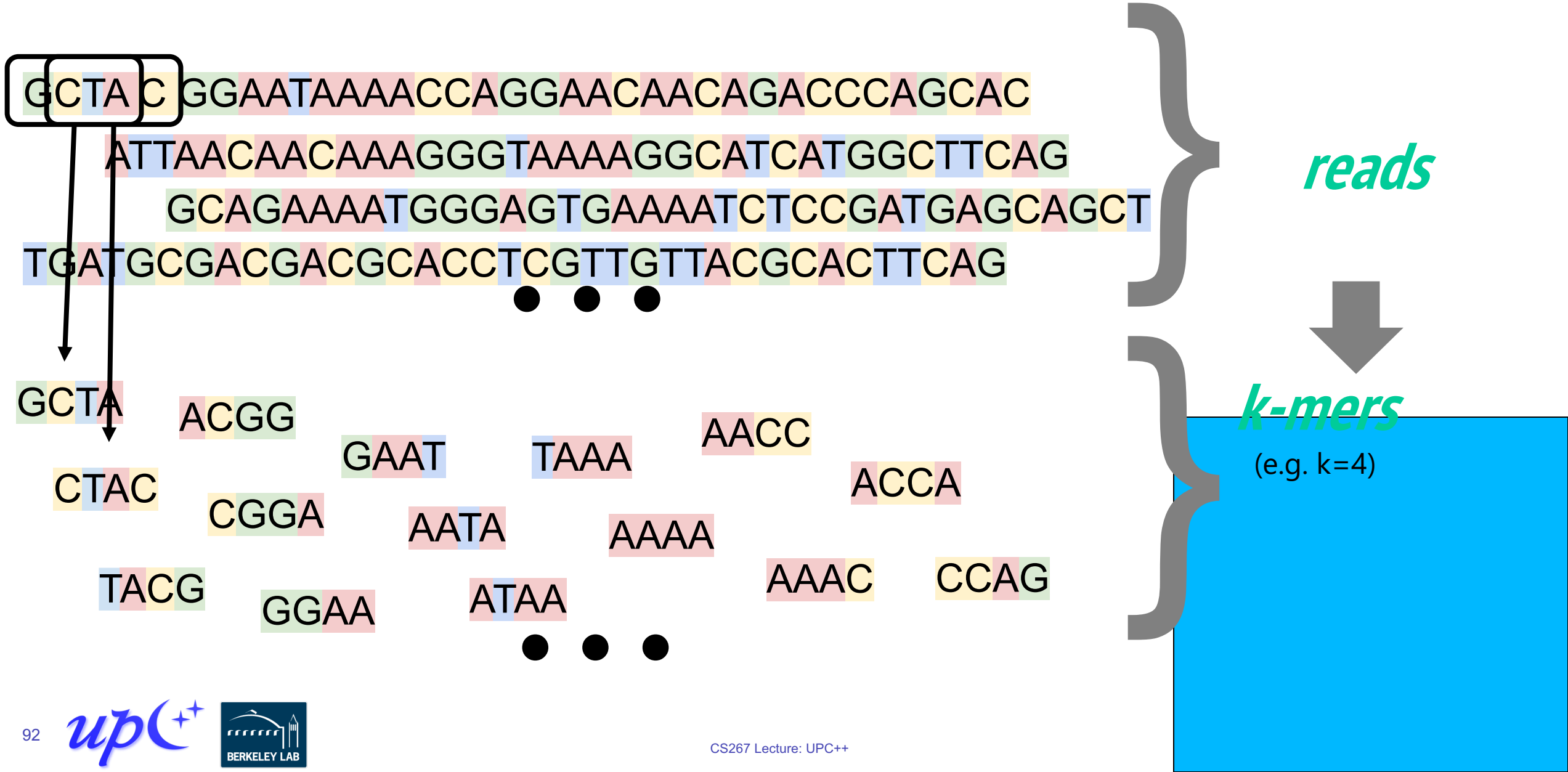
4) Scaffolding

Graph walk

Originally written in MPI & UPC, now in UPC++

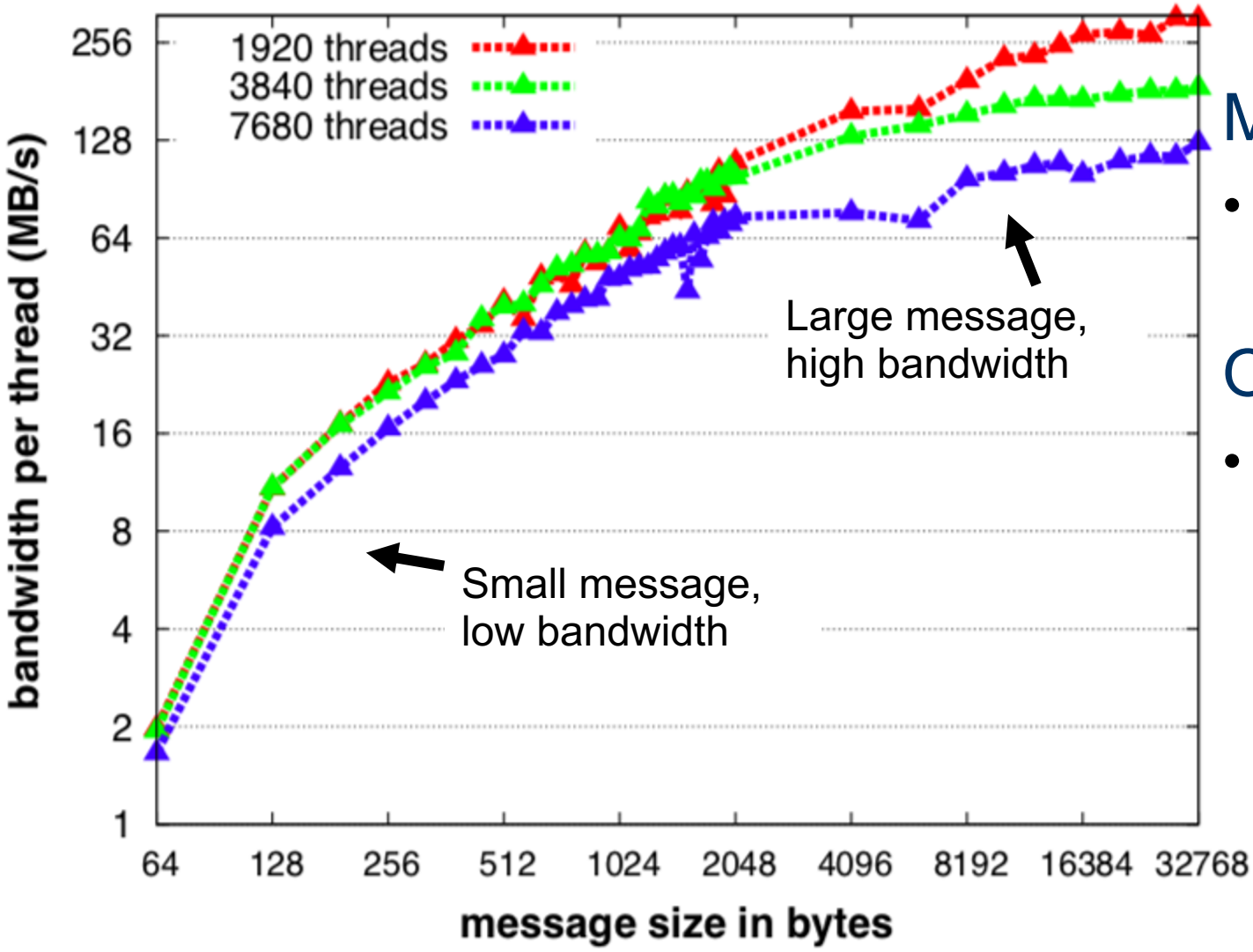
Steve Hofmeyr, Rob Egan, Evangelos Gerganas, leads on MetaHipMer software

K-Mer Analysis Uses a Distributed Hash Table and Bloom Filter



Parallel De Bruijn Graph Construction

ExaBiome / MetaHipMer distributed hashmap



Memory-limited graph stages

- k-mers, contig, scaffolding

Optimized graph construction

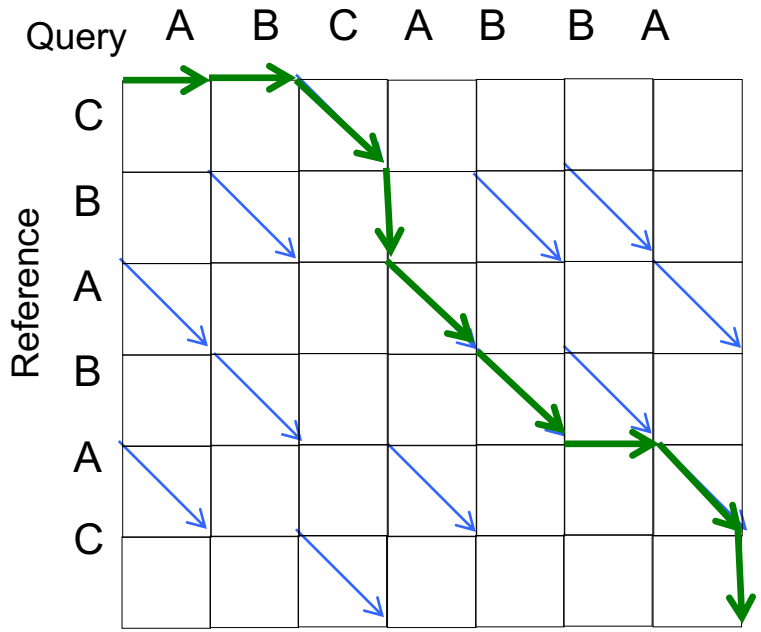
- Dynamically aggregate messages for better effective network bandwidth

Work and results by Rob Egan, ECP ExaBiome Group

Distributed Alignment: Hash Tables and Alignment

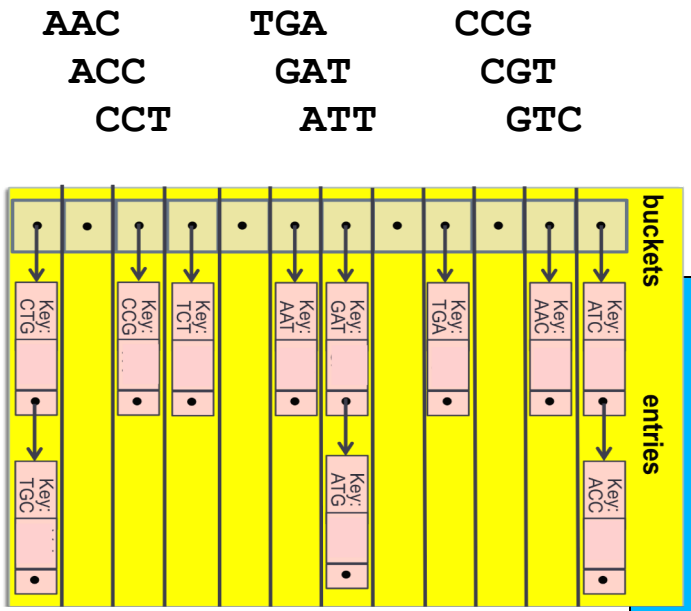
Given strings s and t, align to find minimum # of edits

Dynamic programming on short strings with early stopping for bad alignments



Given sets of strings S and T, find good alignments

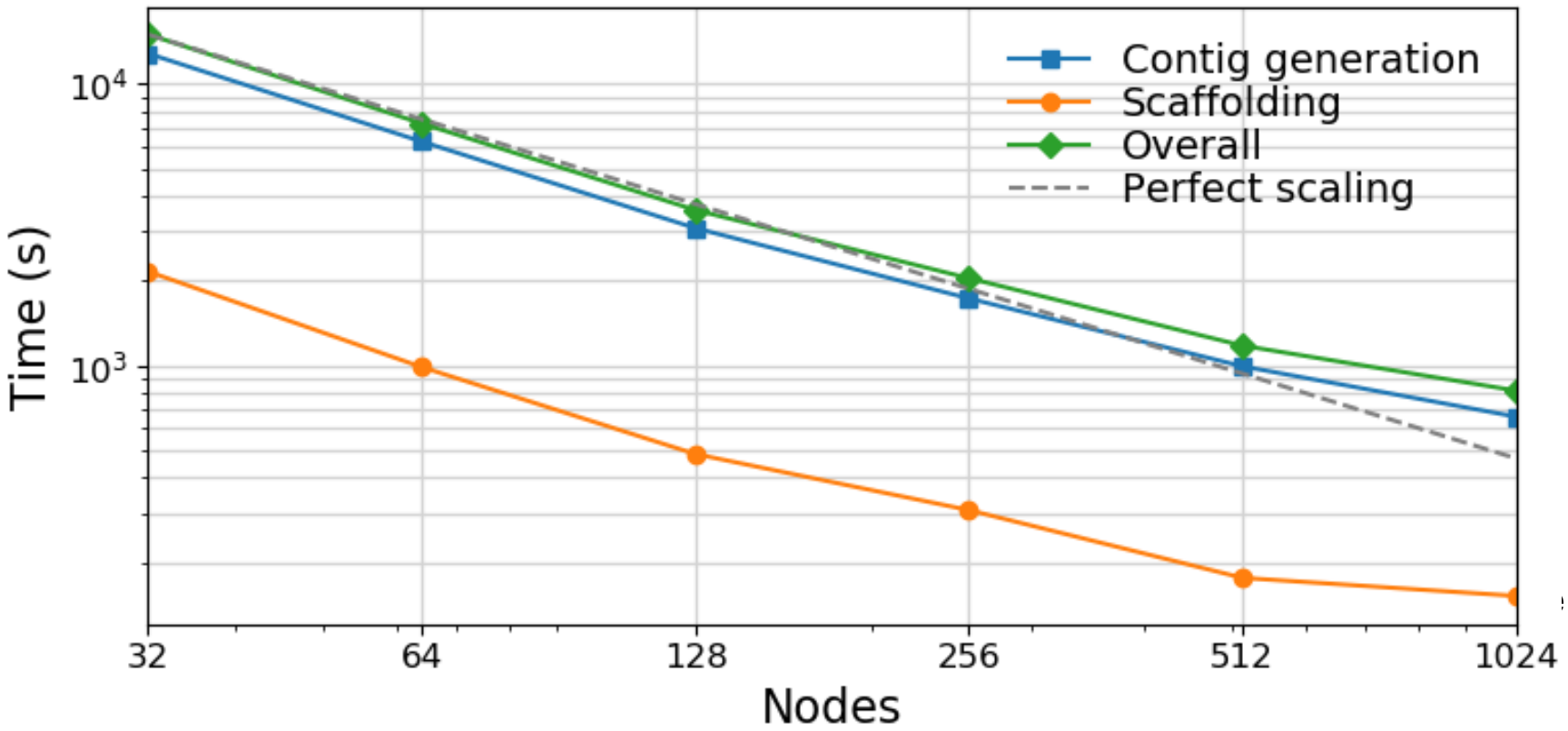
Make hash table of k-mers in S, only align to things in T with at least 1 identical k-mer



1-sided comm or irregular all-to-all + memory

Many variations of both!

MetaHipMer Scaling



Open source: https://sites.google.com/lbl.gov/exabiome/downloads

Runs without errors on several datasets and on multiple HPC systems.

The quality is comparable to other metagenome assemblers

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