

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

Katherine A. Yelick, Amir Kamil, Damian Rouson

<https://go.lbl.gov/sc21>
pagoda@lbl.gov



GASNet-EX

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



Acknowledgements

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

Hadia Ahmed, John Bachan, Scott B. Baden, Dan Bonachea, Rob Egan, Max Grossman, Paul H. Hargrove, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Damian Rouson, Erich Strohmaier, Daniel Waters, Katherine Yelick

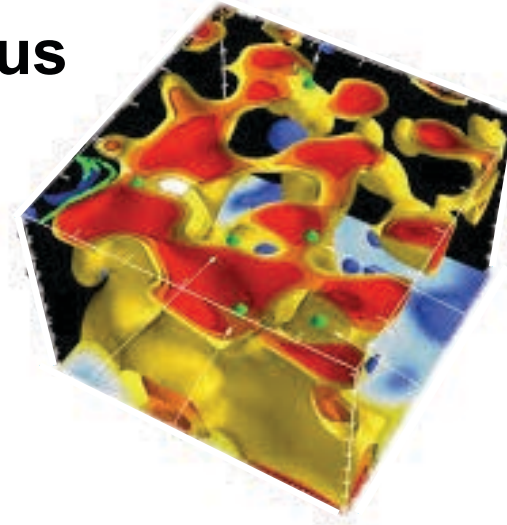
This research was supported in part by the **Exascale Computing Project** (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

This research used resources of the **National Energy Research Scientific Computing Center (NERSC)**, a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231, as well as This research used resources of the **Oak Ridge Leadership Computing Facility** at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

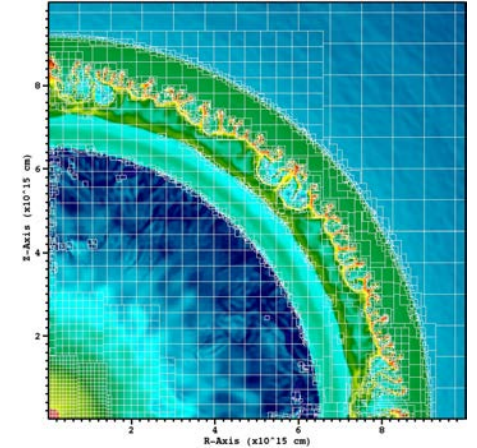
Some motivating applications

Many applications involve asynchronous updates to irregular data structures

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



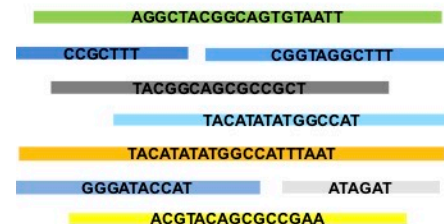
Seismo, Berkeley



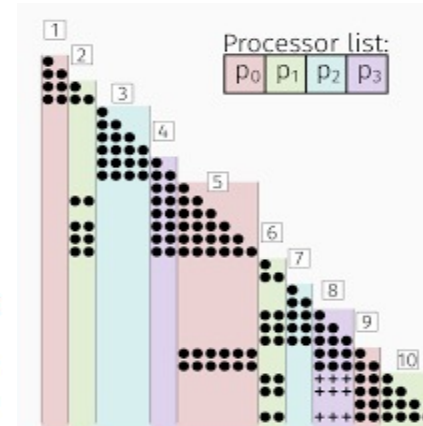
AMReX

Irregular and unpredictable data movement:

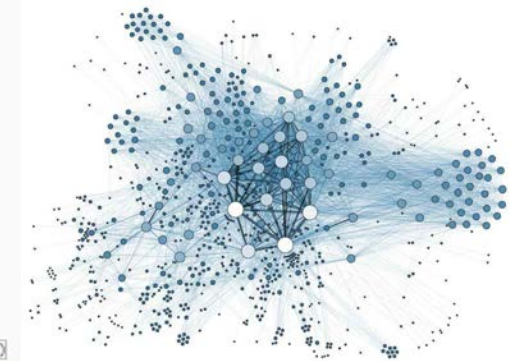
- *Space*: Pattern across processors
- *Time*: When data moves
- *Volume*: Size of data



ExaBiome



SymPACK



Graph analytics

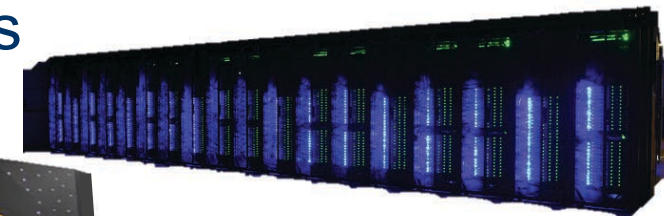
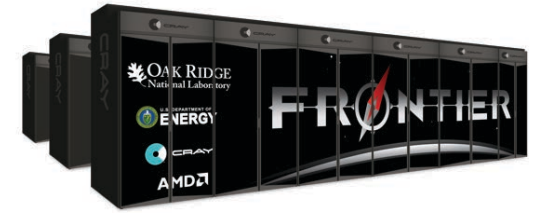
Some motivating system trends

The first exascale systems 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)



Reducing communication overhead

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

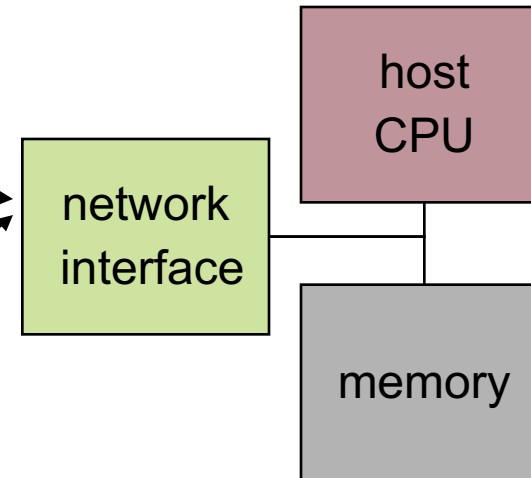
Communication is **one-sided**

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

two-sided message



one-sided put message

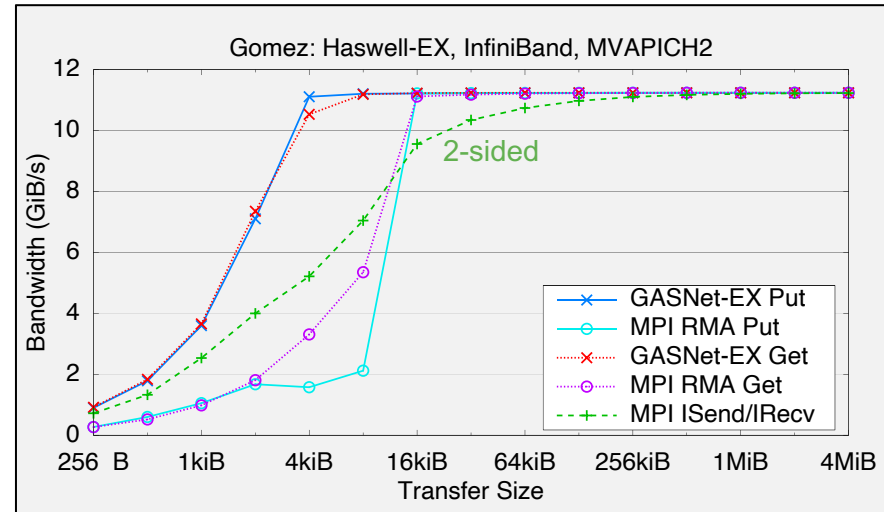
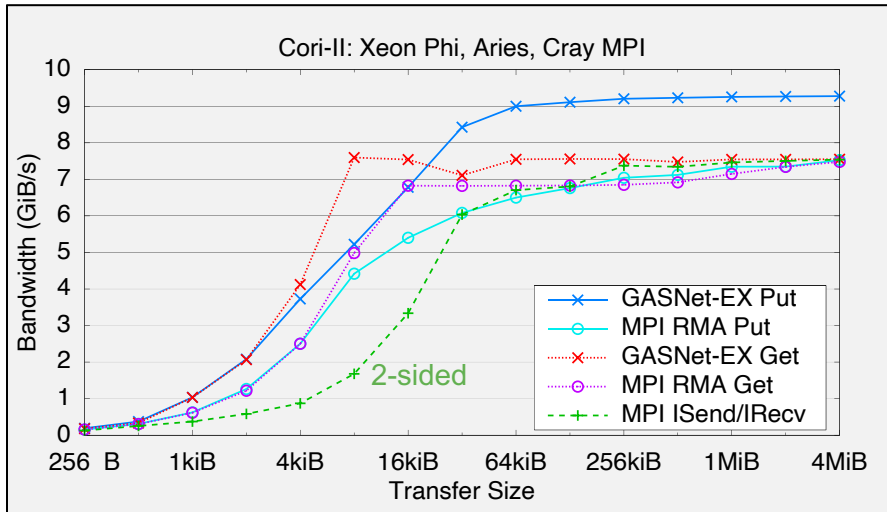
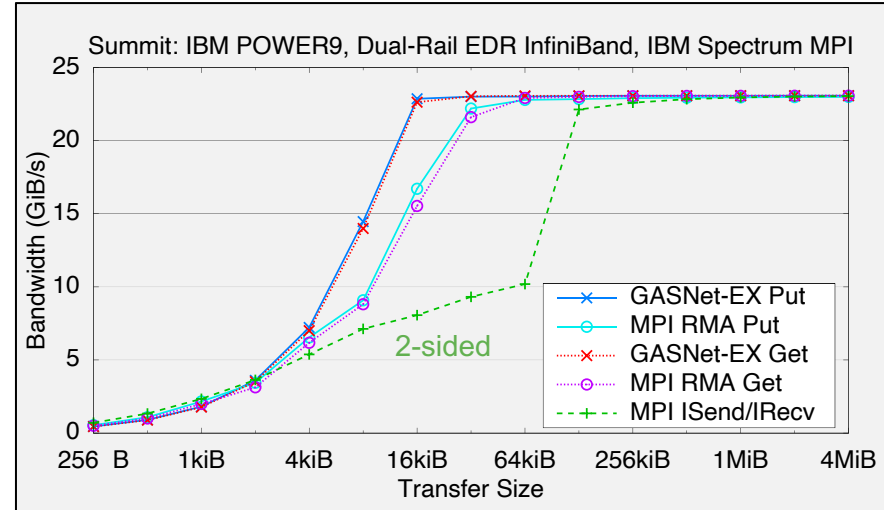
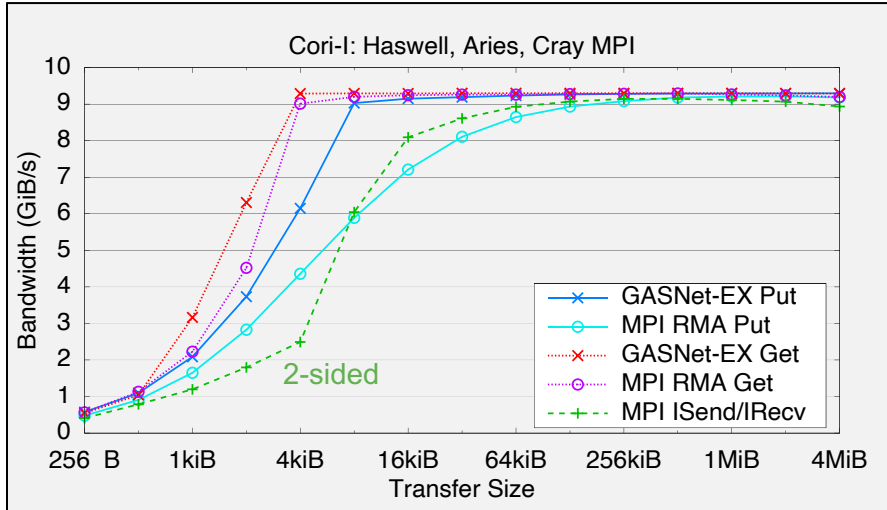
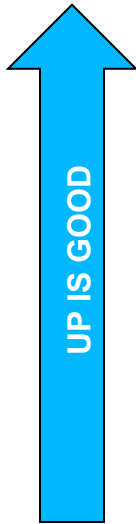


- All metadata provided by the initiator, rather than split between sender and receiver
- Supported in hardware through RDMA (Remote Direct Memory Access)

Looks like shared memory: shared data structures with asynchronous access

One-sided vs Two-sided Message Performance

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



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

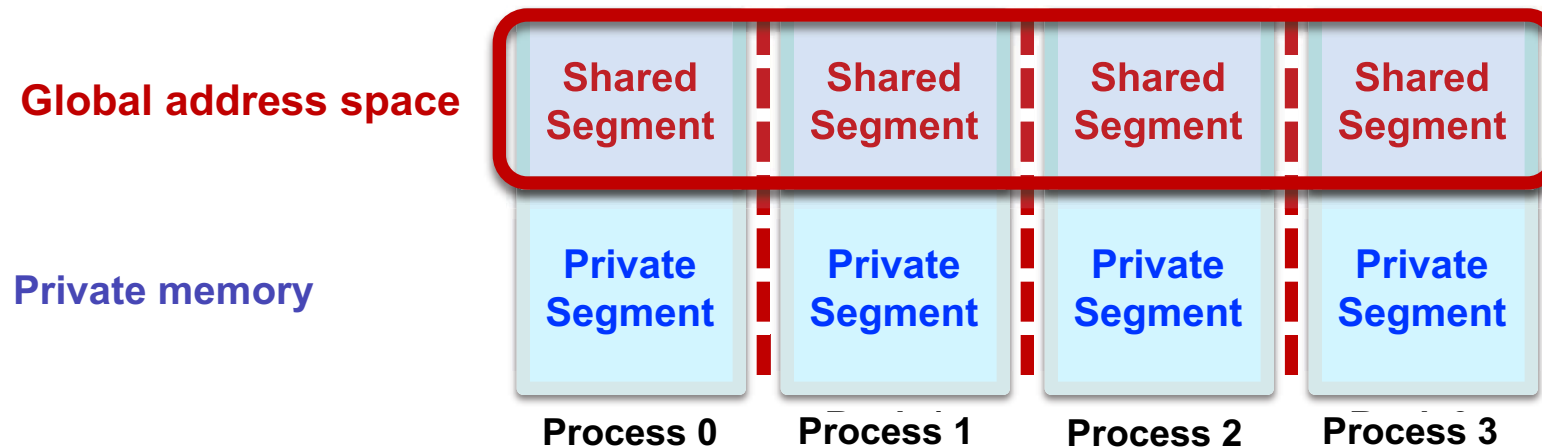
A Partitioned Global Address Space programming model

Global Address Space

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

Partitioned

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



The PGAS model

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

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

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

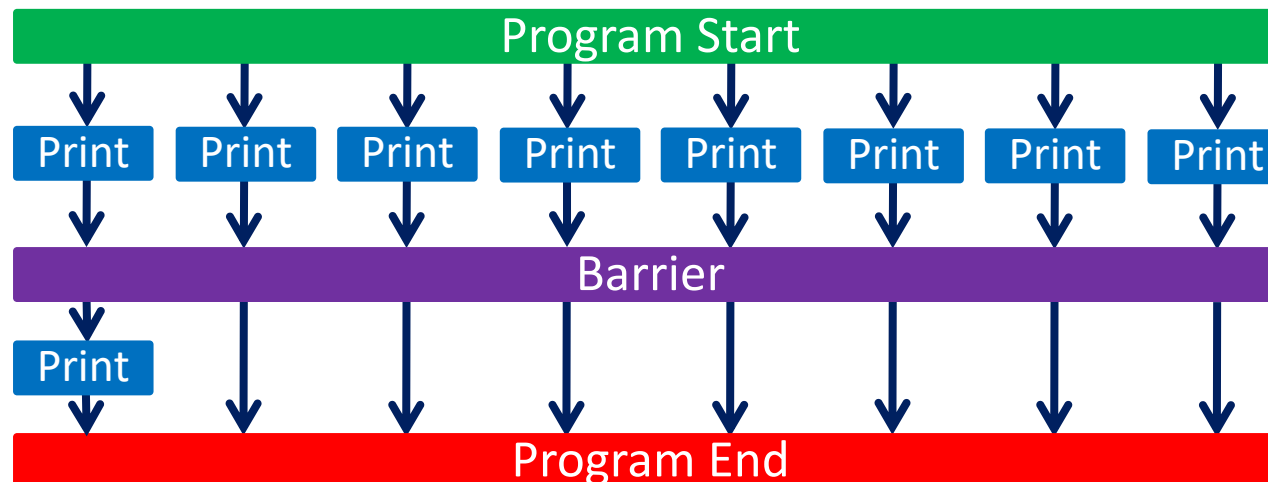
Libraries that provide PGAS: Habanero UPC++, OpenSHMEM, Co-Array C++, Global Arrays, DASH, MPI-RMA

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

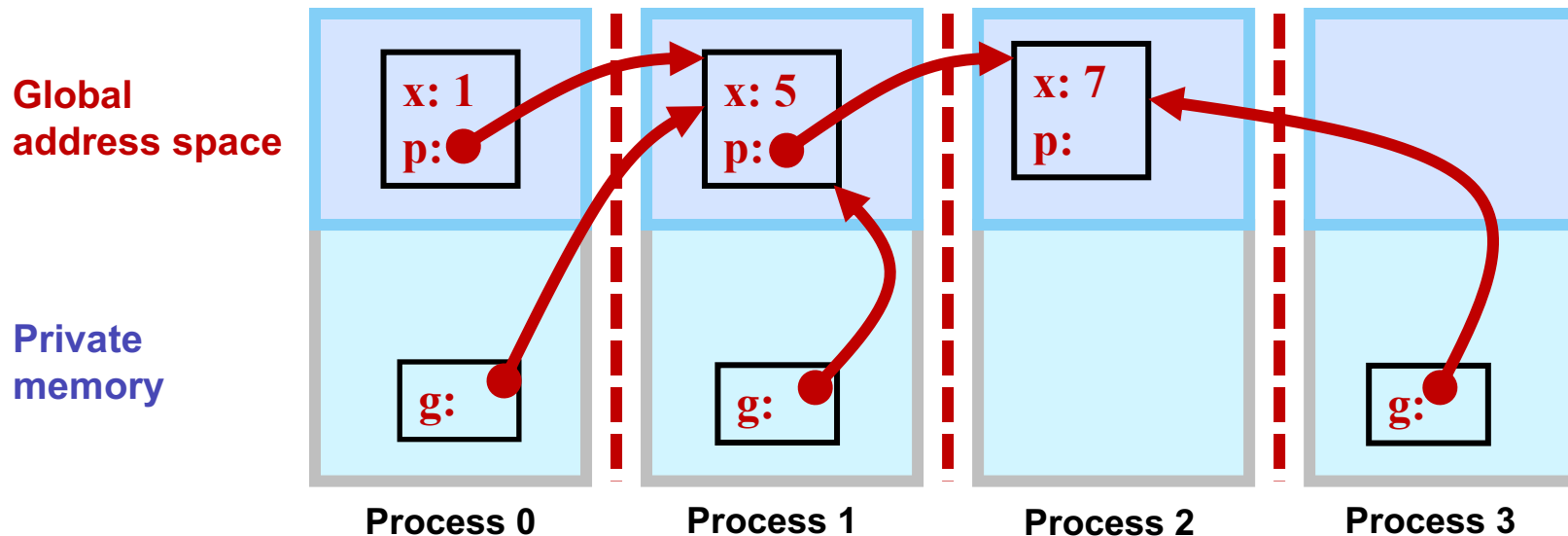
```
int main() {  
    upcxx::init();  
    cout << "Hello from " << upcxx::rank_me() << endl;  
    upcxx::barrier();  
    if (upcxx::rank_me() == 0) cout << "Done." << endl;  
    upcxx::finalize();  
}
```



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>`

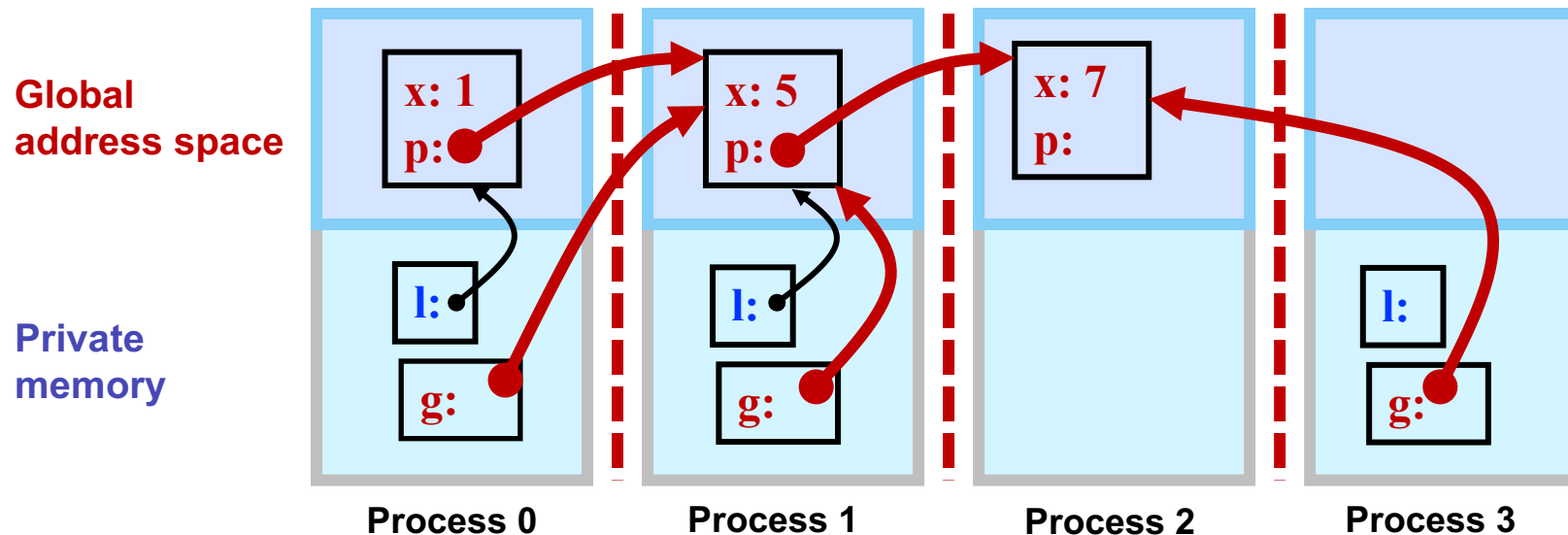


Global vs raw pointers and affinity

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



How does UPC++ deliver the PGAS model?

UPC++ uses a “Compiler-Free,” library approach

- UPC++ leverages C++ standards, needs only a standard C++ compiler



Relies on GASNet-EX for low-overhead communication

- Efficiently utilizes network hardware, including RDMA
- Provides Active Messages on which 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

RMA performance: GASNet-EX vs MPI-3

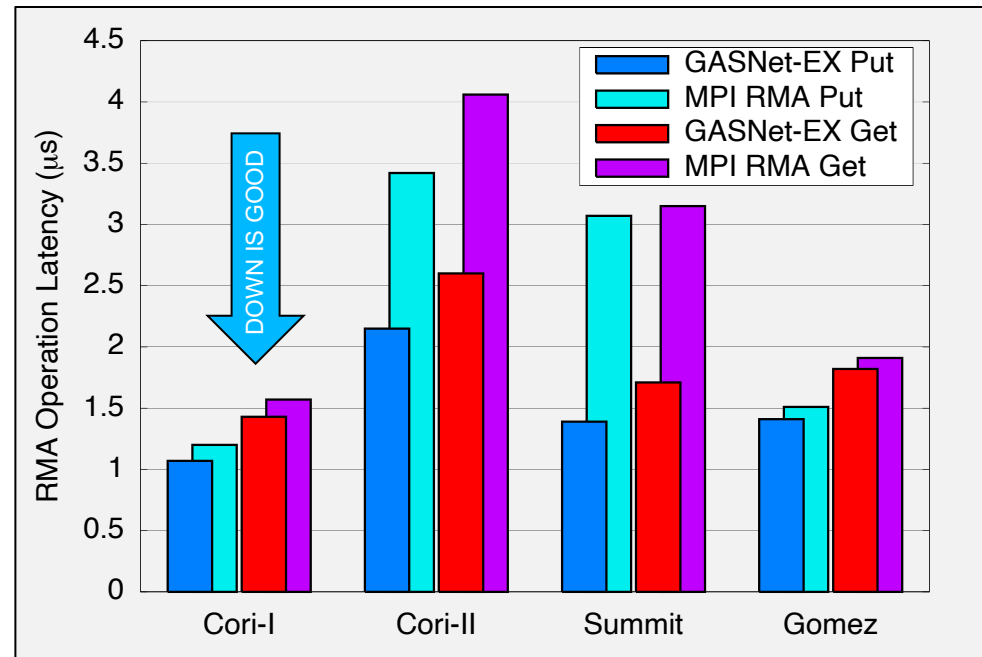
Three different MPI implementations

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 $\frac{1}{2}$ or $\frac{1}{4}$ the transfer size (next slide)

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

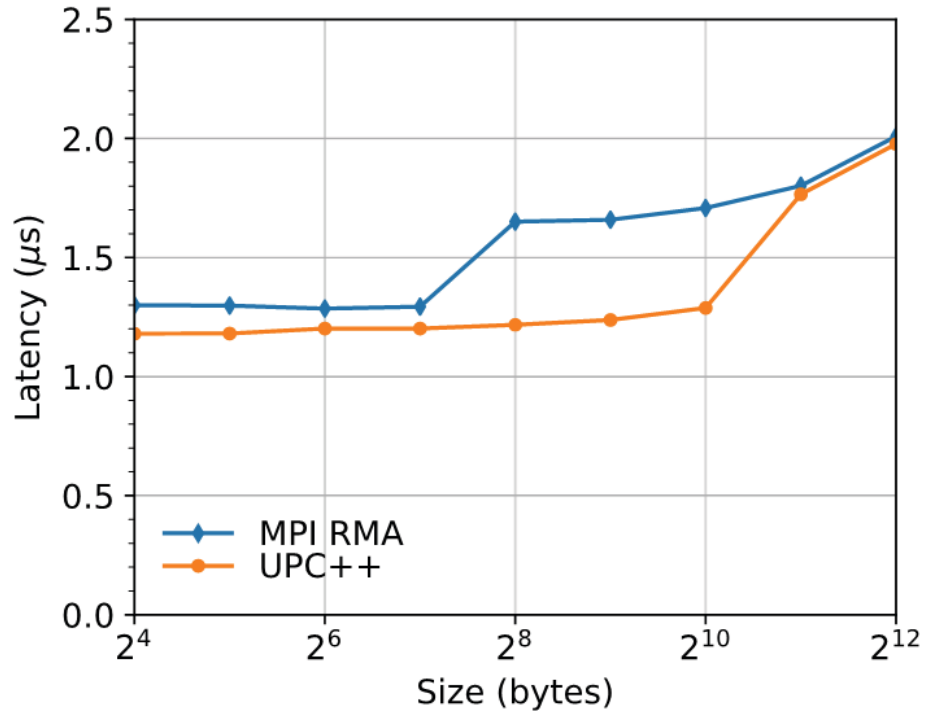


GASNet-EX results from v2018.9.0 and v2019.6.0. MPI results from Intel MPI Benchmarks v2018.1. For more details see Languages and Compilers for Parallel Computing (LCPC'18). <https://doi.org/10.25344/S4QP4W>
More recent results on Summit here replace the paper's results from the older Summitdev.

UPC++ on top of GASNet

Experiments on NERSC Cori:

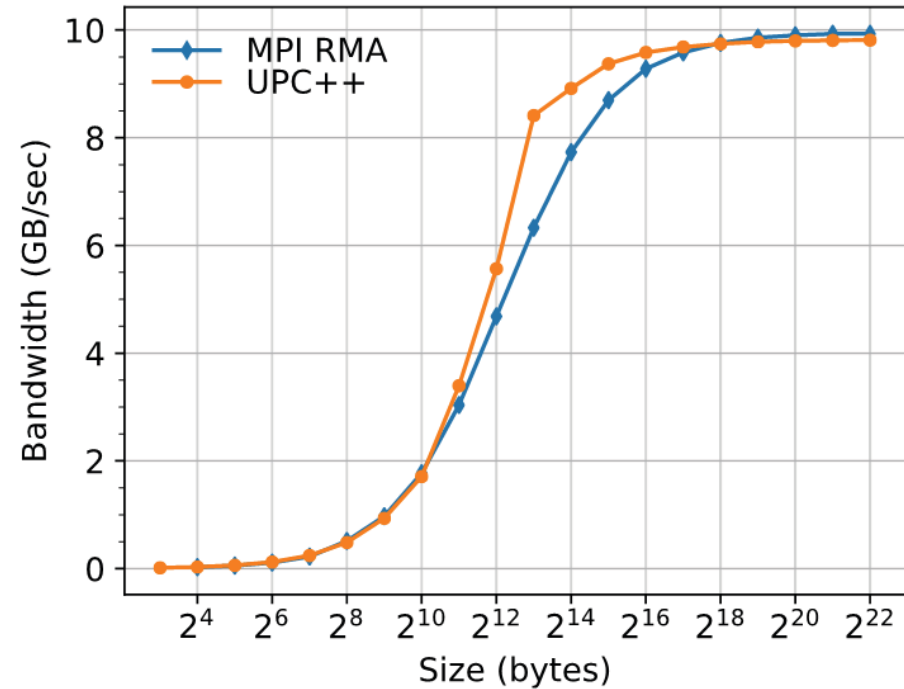
- Cray XC40 system



Round-trip Put Latency (lower is better)

Two processor partitions:

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



Flood Put Bandwidth (higher is better)

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

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)

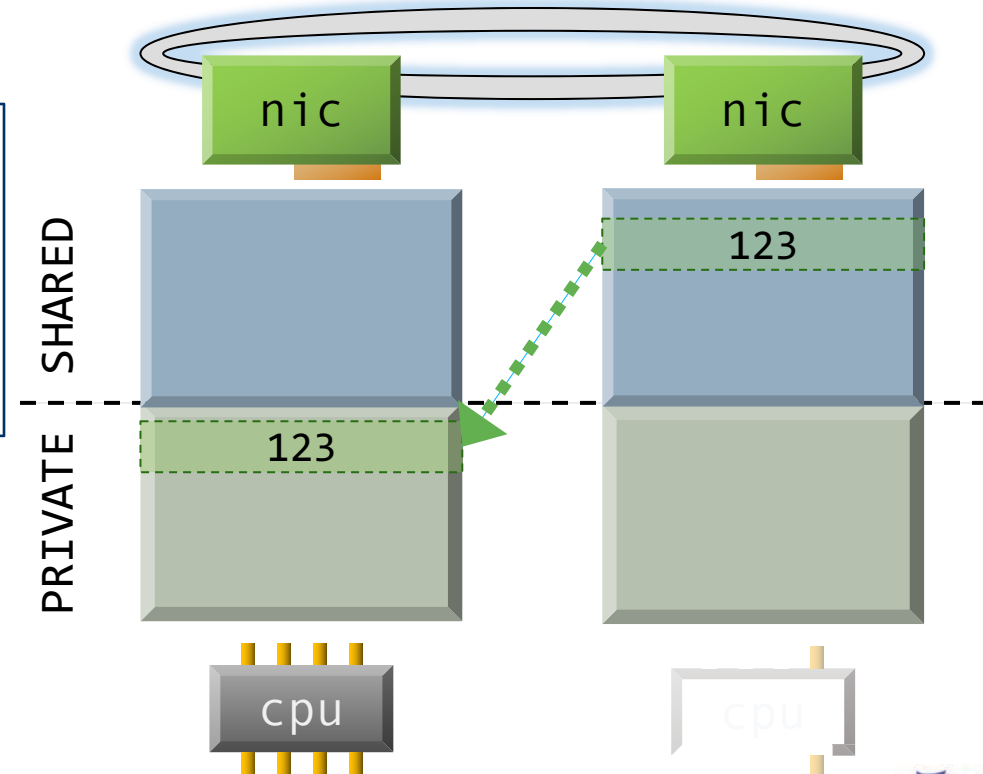
By default, all communication operations are split-phased

- **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

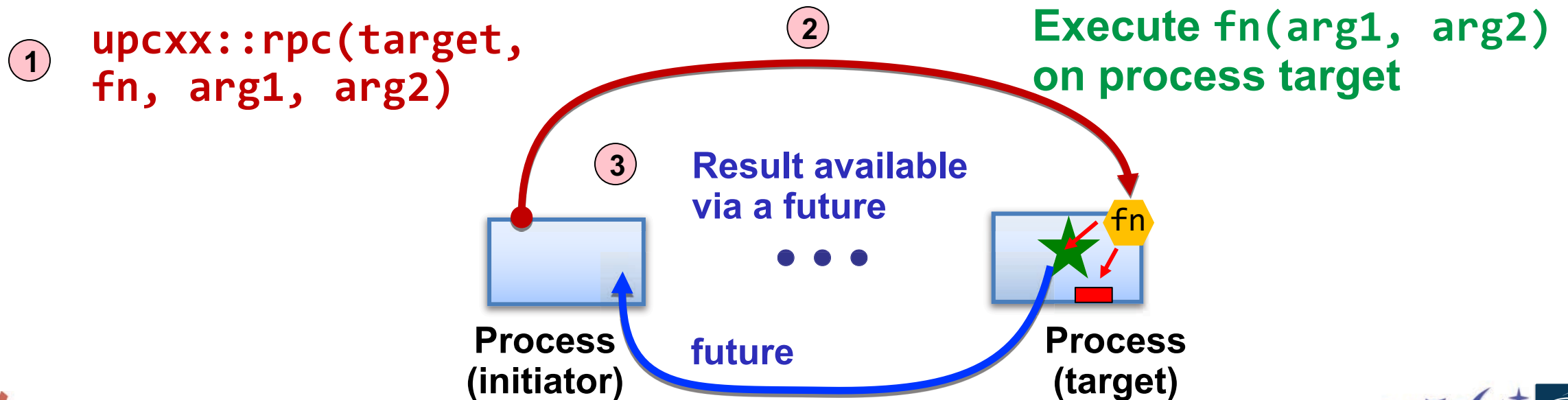


Remote procedure call (RPC)

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

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

Many RPCs can be active simultaneously, hiding latency



Compiling and running a UPC++ program

UPC++ provides tools for ease-of-use

Compiler wrapper:

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

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

Launch wrapper:

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

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

Using UPC++ at US DOE Office of Science Centers

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
<https://go.lbl.gov/sc21>

Also contains links to source, build instructions, and a docker image
UPC++ works on laptops, workstations and clusters too.

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
```

Exercise 0: Hello world compile and run

Everything needed for the hands-on activities is at:

<https://go.lbl.gov/sc21>

Online materials include:

- Module info for NERSC Cori, OLCF Summit and ALCF Theta
- Download links to install UPC++
 - natively or w/Docker container on your own system

Once you have set up your environment and copied the tutorial materials:

```
elvis@cori07:~> cd 2021-11/exercises/  
elvis@cori07:~/2021-11/exercises> make run-ex0  
[...full path...]/bin/upcxx ex0.cpp -o ex0  
[...full path...]/bin/upcxx-run -n 4 ./ex0  
Hello world from process 2 out of 4 processes  
Hello world from process 0 out of 4 processes  
Hello world from process 3 out of 4 processes  
Hello world from process 1 out of 4 processes
```

Exercise 1: Ordered hello world

Modify the program below so that the messages are written to the output file in order by rank (ex1.cpp)

- Processes should take turns printing to the file, using a loop in which one process prints per iteration
- Use `upcxx::barrier()` to perform a *barrier*, which prevents any process from continuing until all processes have reached it

```
int main() {  
    upcxx::init();  
    std::ofstream fout("output.txt", std::iosbase::app);  
    fout << "Hello from process " << upcxx::rank_me()  
          << " out of " << upcxx::rank_n() << std::endl;  
    sync();  
    upcxx::finalize();  
}
```

Commit data to disk
(POSIX systems)

[Link to solution](#)

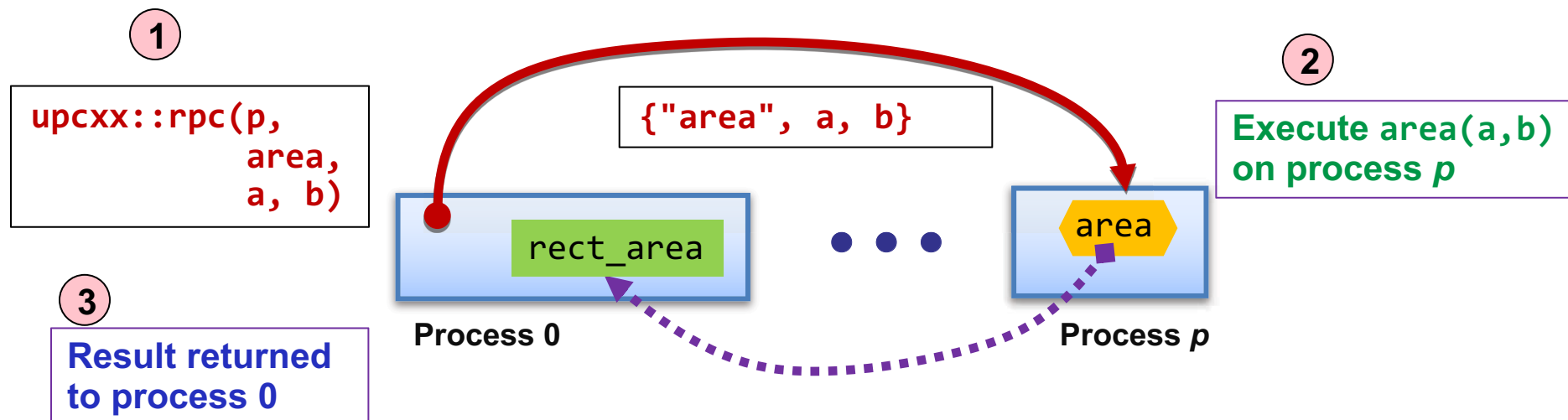
Remote Procedure Calls (RPC)

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

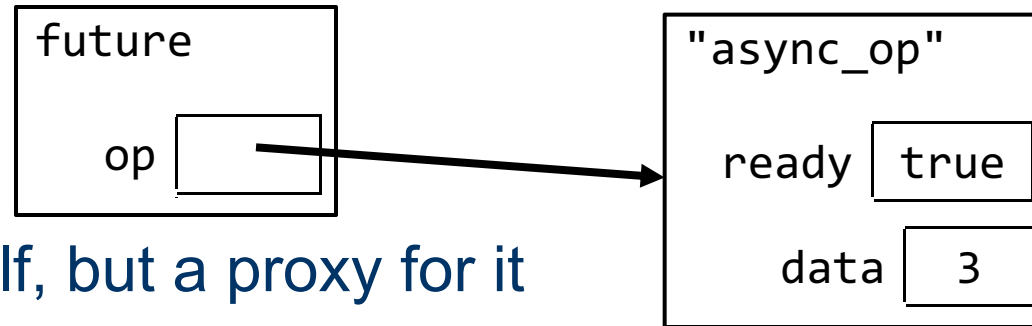
```
upcxx::future<> fut =  
    upcxx::rpc(0, [](int rank) {  
        cout << "Hello from process " << rank << endl;  
    }, upcxx::rank_me());  
  
fut.wait();
```

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

What is a future?

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

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



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

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

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

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

Overlapping communication

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

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

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

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

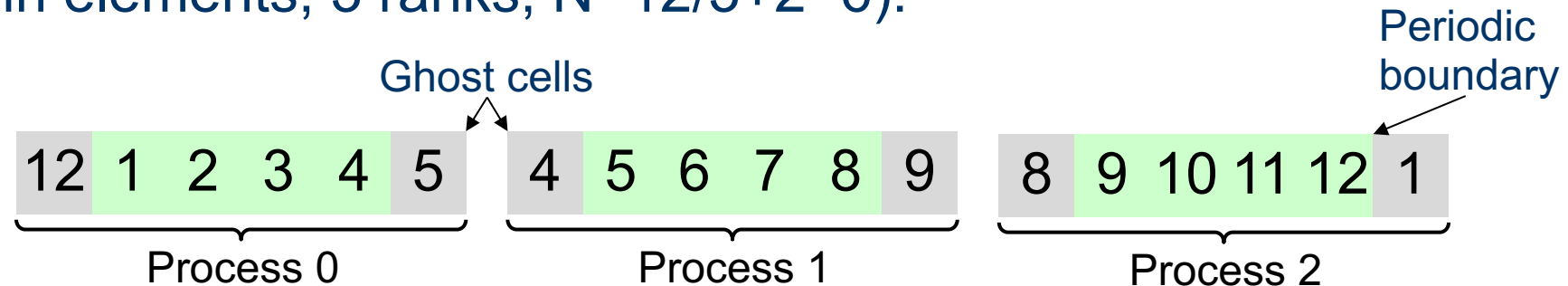
1D 3-point Jacobi in UPC++

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

Out-of-place computation requires two grids **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

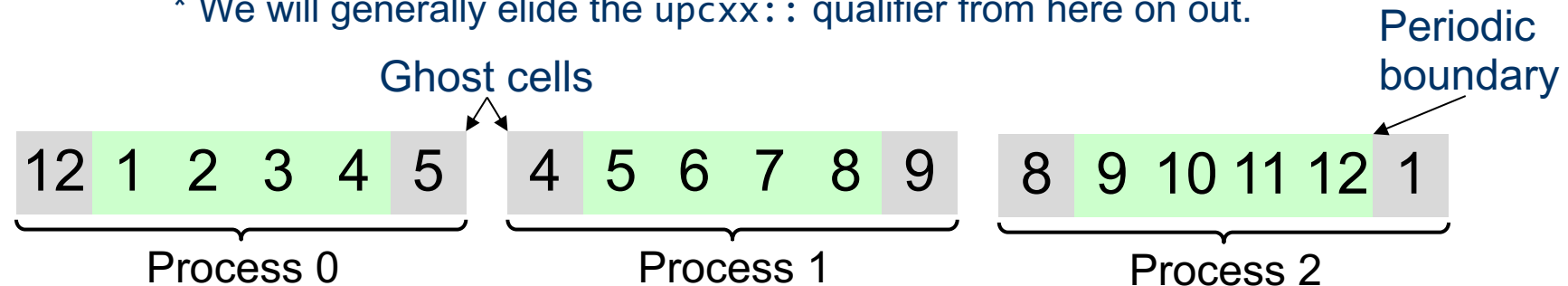
```
double *old_grid, *new_grid;
```

```
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)

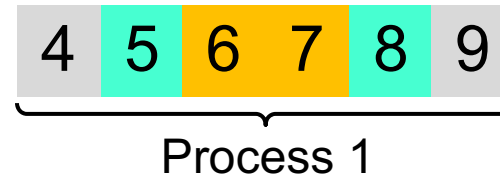
We can use RPC to communicate boundary cells

```
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);
```

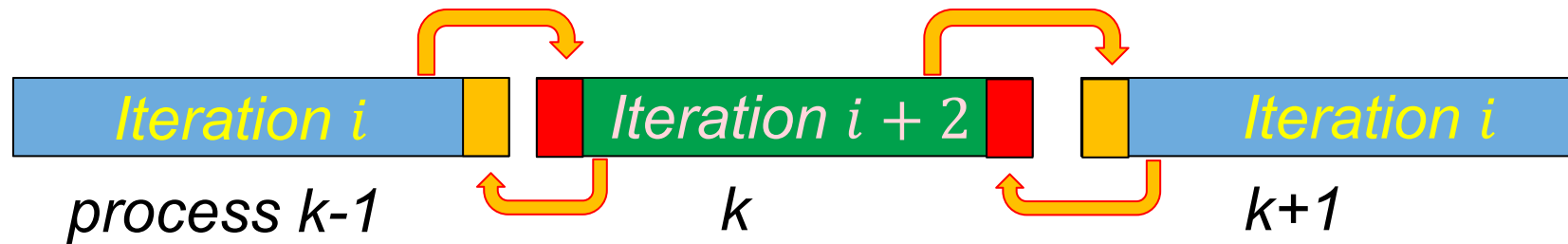


Race conditions

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:

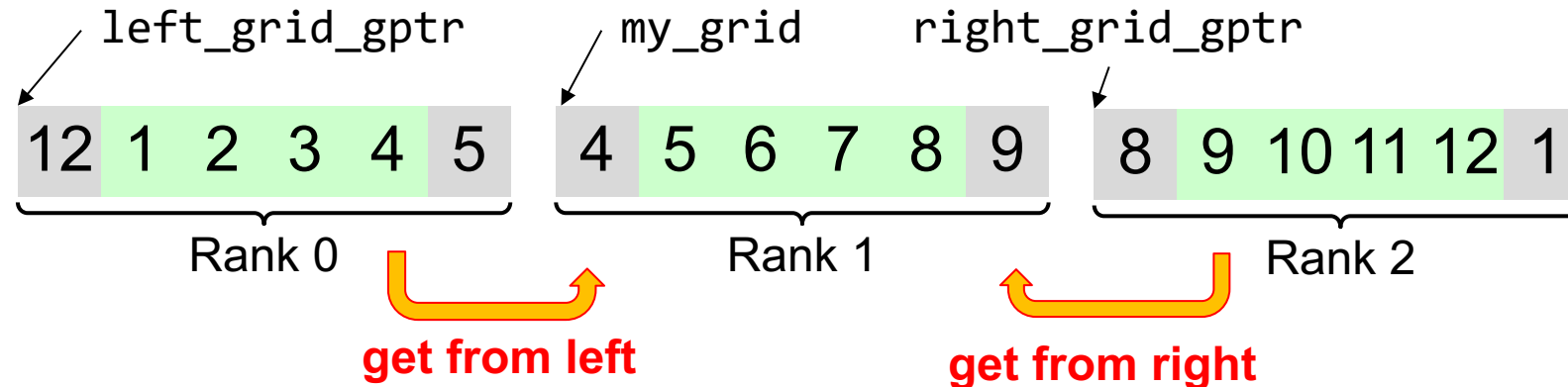
```
global_ptr<int> remote = /* ... */;  
future<int> fut1 = rget(remote);  
int result = fut1.wait();  
future<> fut2 = rput(42, remote);  
fut2.wait();
```

- 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 to optimize for co-located 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);
```

Remote source (global_ptr) Local dest ptr

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

Overlapped computation
on interior cells

Begin asynchronous
RMA gets

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

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

Wait for communication,
then consume values

Callbacks

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

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

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

← Vector get does not produce a value

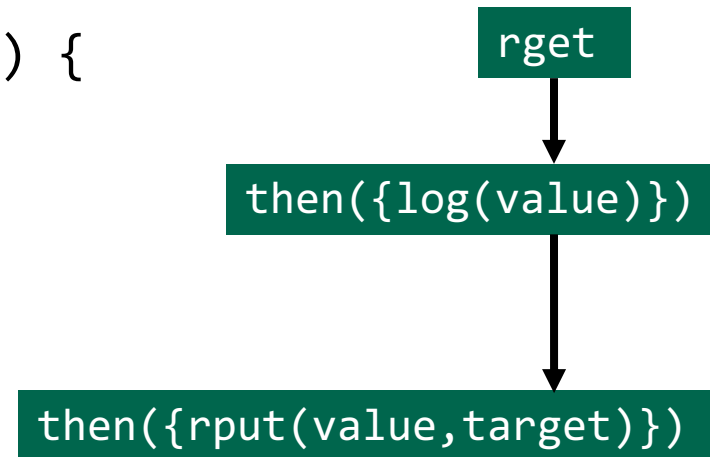
```
future<> right_update =  
  rget(right_old_grid + N - 2)  
  .then([](double value) {  
    new_grid[N-2] = 0.25 *  
      (old_grid[N-3] + 2*old_grid[N-2] + value);  
  });
```

← Scalar get produces a value

Chaining callbacks

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

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



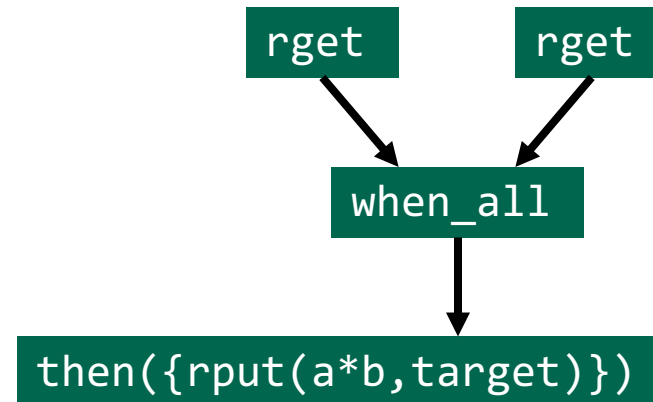
This code retrieves an integer from a remote location, computes its log, and then sends it to a different remote location

Conjoining futures

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

Can be used to specify multiple dependencies for a callback

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



Jacobi RMA with puts and conjoining

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

```
future<> puts = when_all(  
    rput(old_grid[1], left_old_grid + N - 1),  
    rput(old_grid[N-2], right_old_grid));  
  
for (long i = 2; i < N - 2; ++i)  
    /* ... */;
```

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

Ensure outgoing puts have completed

Ensure incoming puts have completed

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

Distributed objects

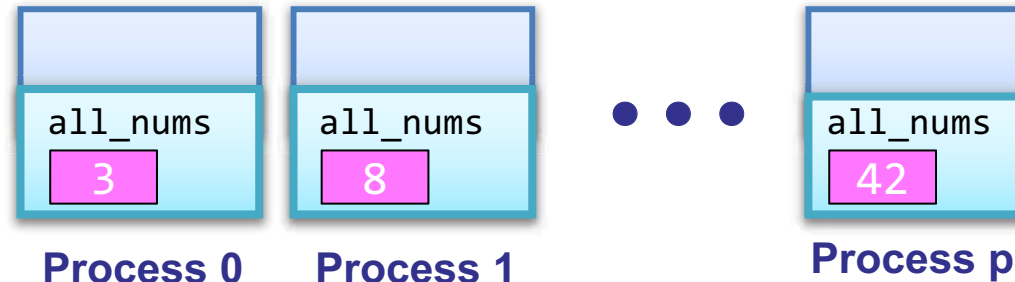
A *distributed object* is an object that is partitioned over a set of processes

```
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());
```

Example: Monte Carlo computation of pi

Estimate pi by throwing darts at a unit square

Calculate percentage that fall in the unit circle

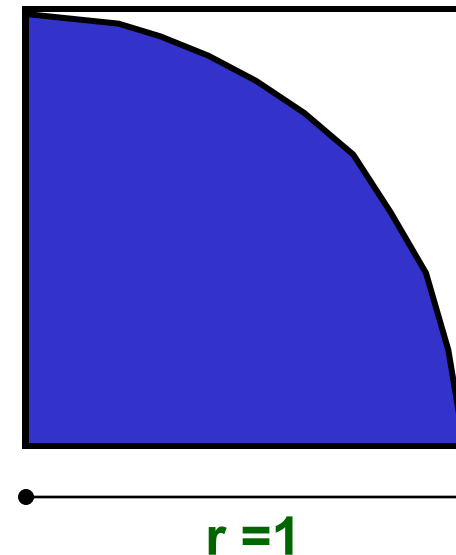
- Area of square = $r^2 = 1$
- Area of circle quadrant = $\frac{1}{4} * \pi r^2 = \pi/4$

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 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

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 initiator and the target

```
int my_hits = 0;
for (int i = 0; i < my_trials; ++i)
    my_hits += hit();
dist_object<int> all_hits(my_hits);
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;
}
```

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

Exercise 2: Distributed object in Jacobi

Modify the Jacobi code to perform bootstrapping using UPC++ distributed objects (ex2.cpp)

```
global_ptr<double> old_grid_gptr, new_grid_gptr;  
global_ptr<double> right_old_grid, right_new_grid;  
int right; // rank of my right neighbor
```

```
// Obtains grid pointers from the right neighbor and  
// sets right_old_grid and right_new_grid accordingly.
```

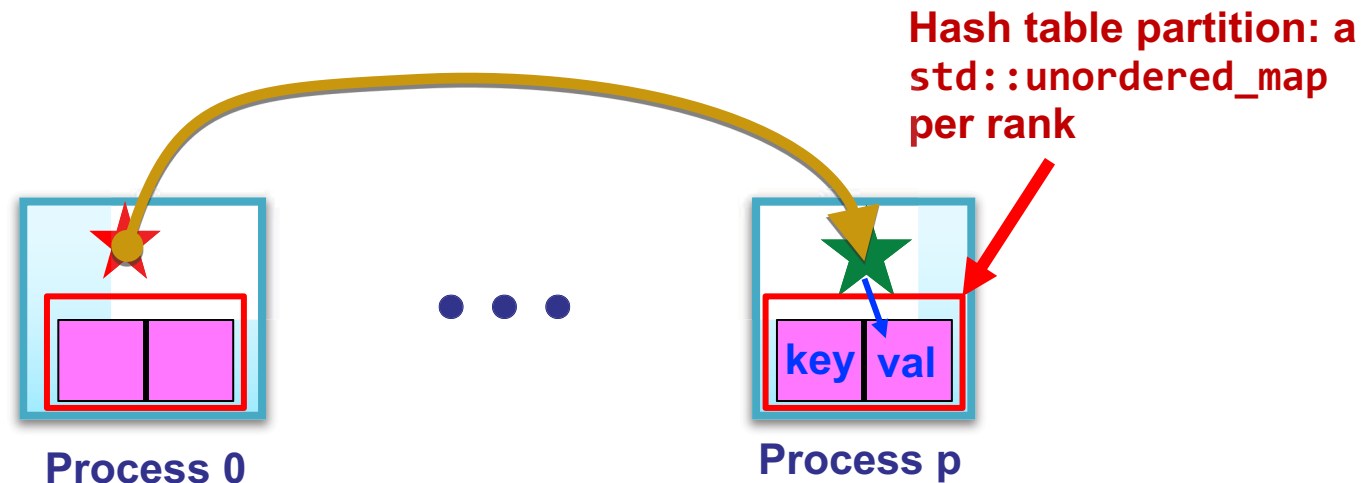
```
void bootstrap_right() {  
  
    /* your code here */  
  
}
```

[Link to solution](#)

Distributed hash table (DHT)

Distributed analog of `std::unordered_map`

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



DHT data representation

A distributed object represents the directory of unordered maps

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

Define an abbreviation for a helper type

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

Computes owner for the given key

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

DHT insertion

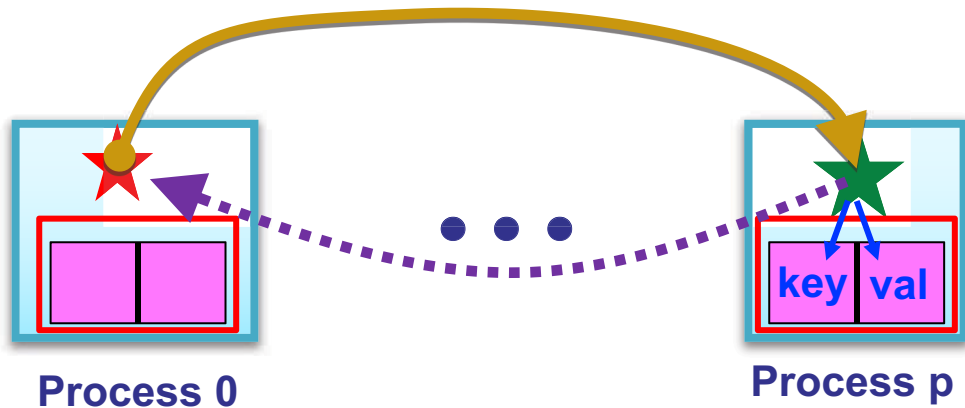
Insertion initiates an RPC to the owner and returns a future that represents completion of the insert

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

Send RPC to the 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),  
        [](doobj_map_t &lmap, const string &key) {  
            if (lmap->count(key) == 0)  
                return string("NOT FOUND");  
            else  
                return (*lmap)[key];  
        }, local_map, key);  
}
```

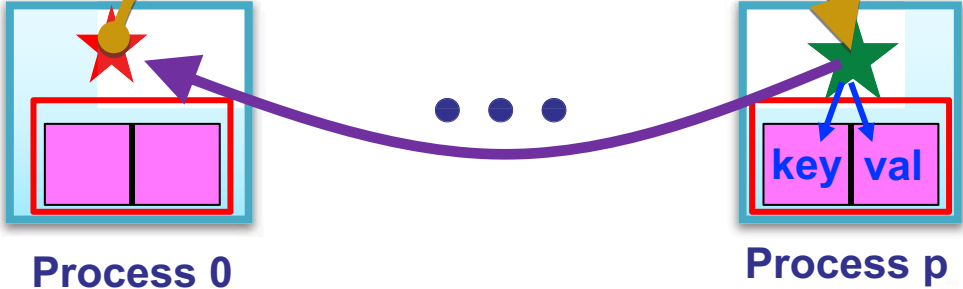
Send RPC to the rank determined by key hash

Check whether key exists in local map

Retrieve corresponding value from the local map and return it

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

Key passed as argument to the remote function



Exercise 3: Distributed hash table

Implement the erase and update methods (ex3.hpp)

```
// Erases the given key from the DHT.
```

```
future<> erase(const string &key);
```

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

```
future<string> update(const string &key, const string &value);
```

```
// Use this function to perform an update on an  
// unordered_map that resides on the local process.  
// Assume it is already written for you.
```

```
static string local_update(unordered_map<string, string> &lmap,  
                           const string &key, const string &value);
```

[Link to solution](#)

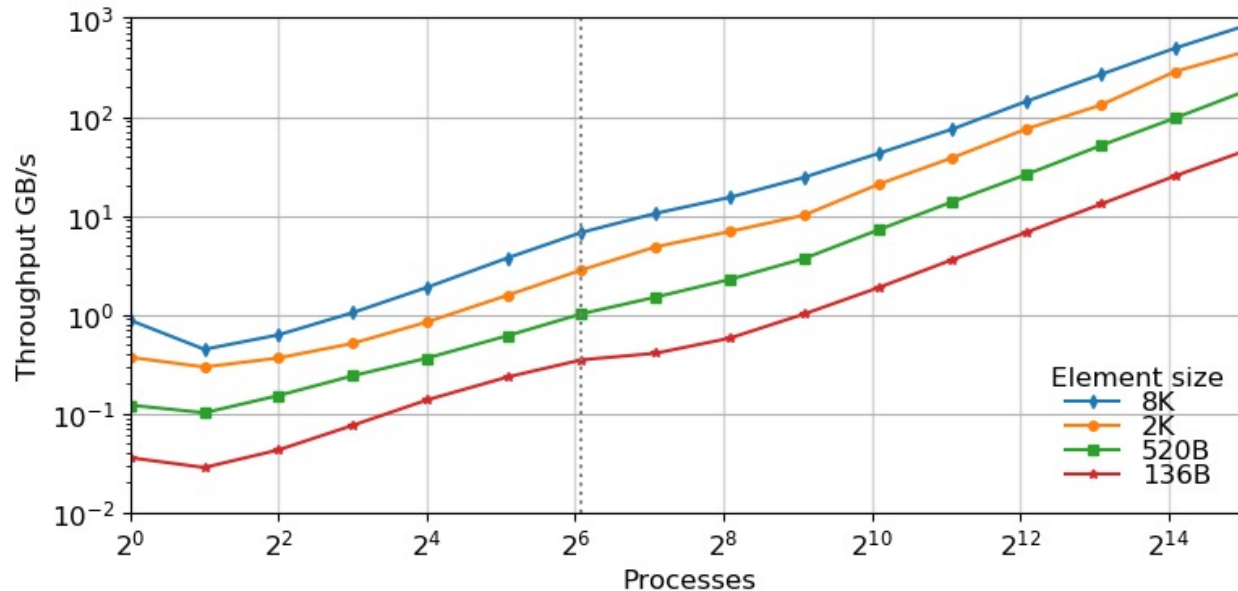
Optimized DHT scales well

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)

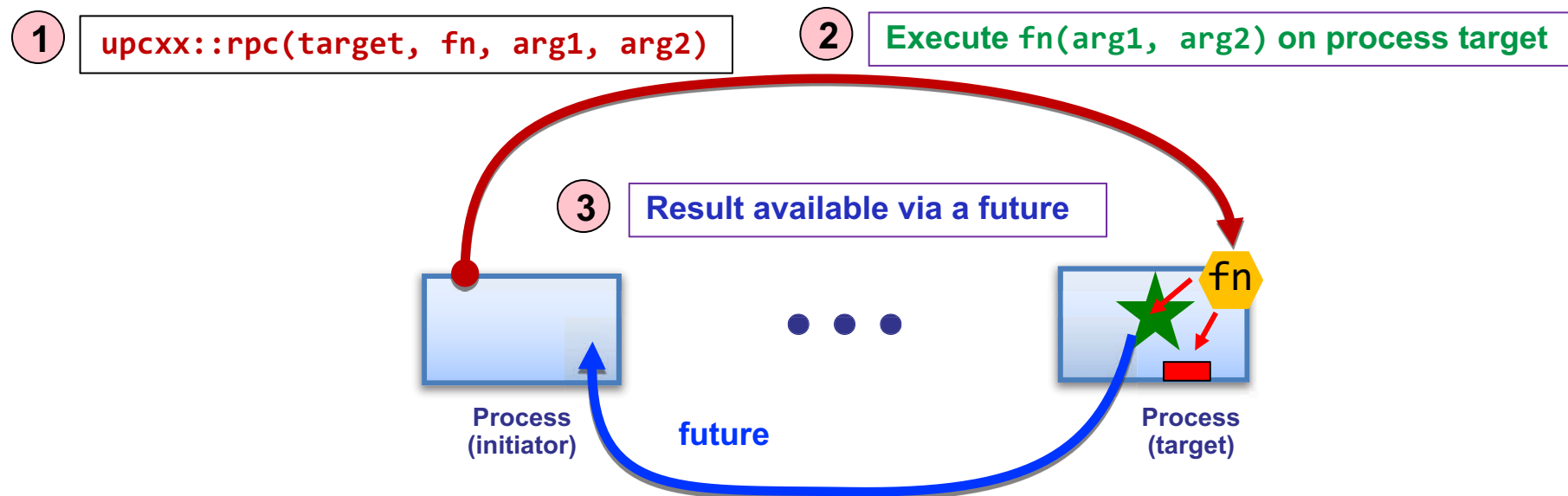
Cray XC40

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 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
 - Readyng futures, running callbacks, invoking inbound RPCs

Invoking user-level progress

The progress() function invokes user-level progress

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

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

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

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

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

Remote atomics

Remote atomic operations are supported with an *atomic domain*

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

The domain dictates the data type and operation set

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

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

Operations are performed on global pointers and are asynchronous

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

Serialization

RPC's transparently *serialize* shipped data

- Conversion between in-memory and byte-stream representations



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

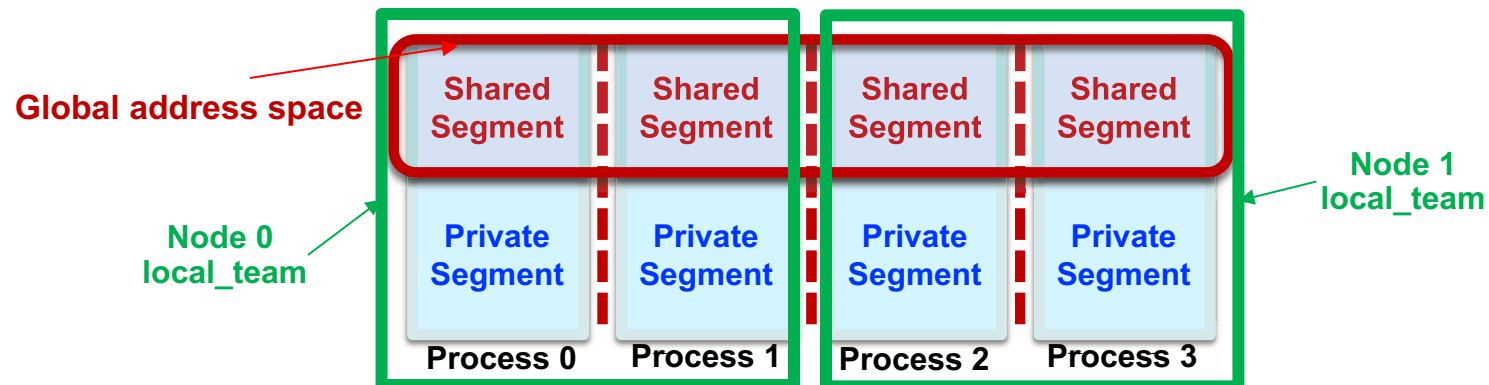
- Some process pairs are “closer” than others
- Ex: cabinet > switch > node > NUMA domain > socket > core

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();
```

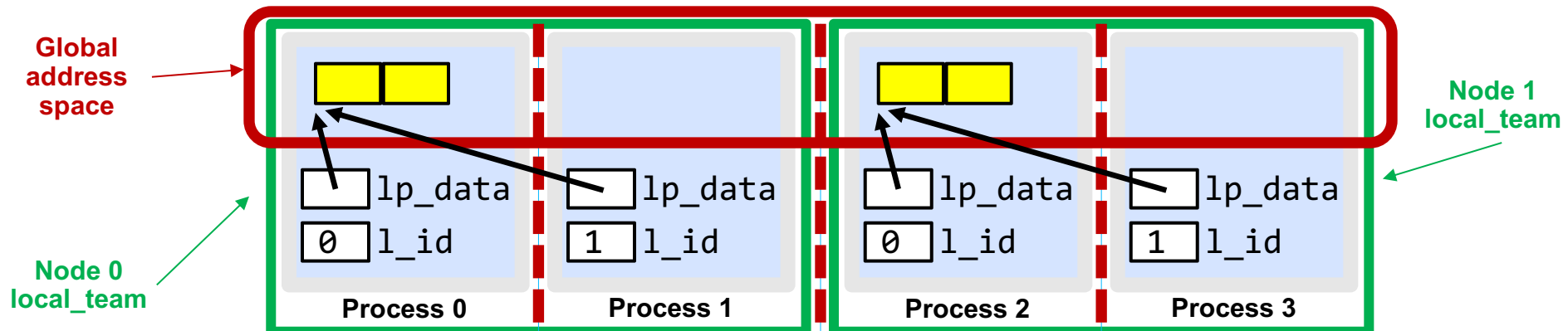
} Allocate and share one array per node

```
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)

Non-contiguous RMA

We've seen contiguous RMA

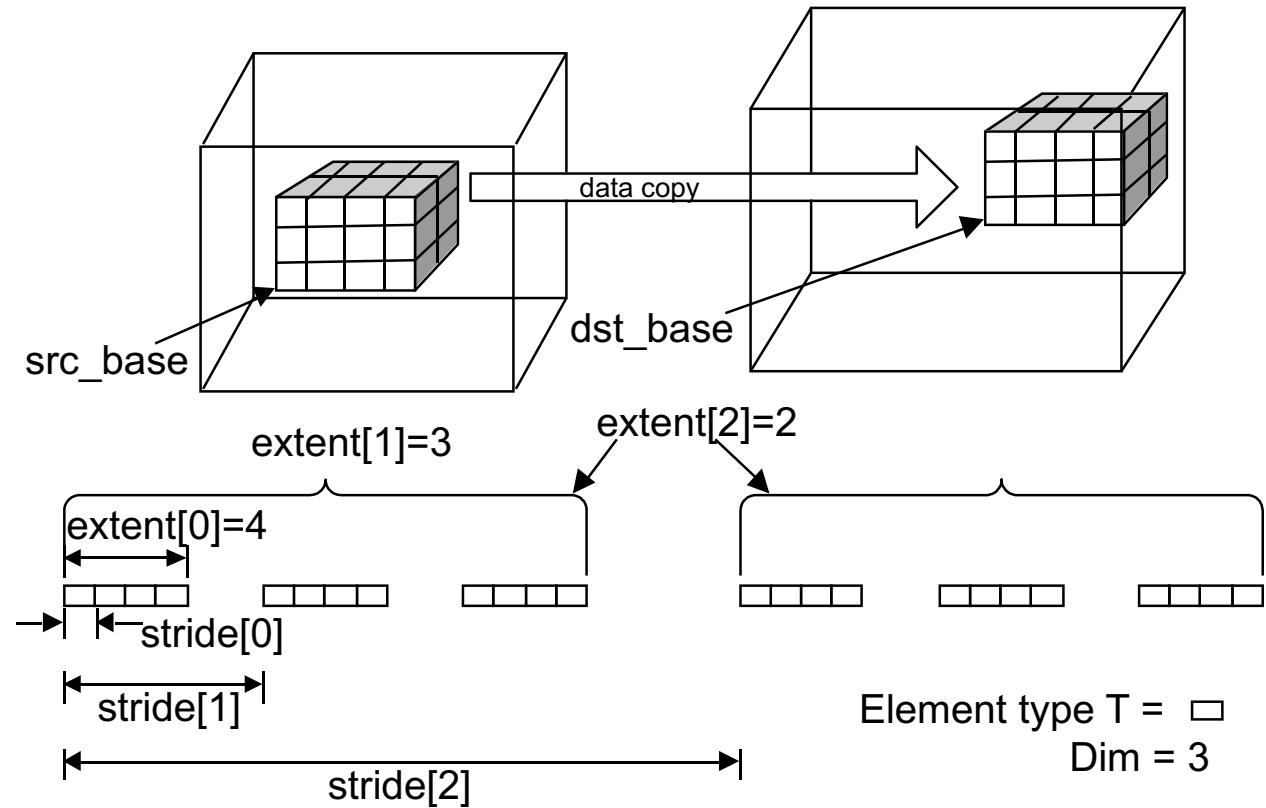
- Single-element
- Dense 1-d array

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



Memory Kinds

Supercomputers are becoming increasingly heterogeneous in compute, memory, storage

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

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

Can point to memory on a local or remote GPU

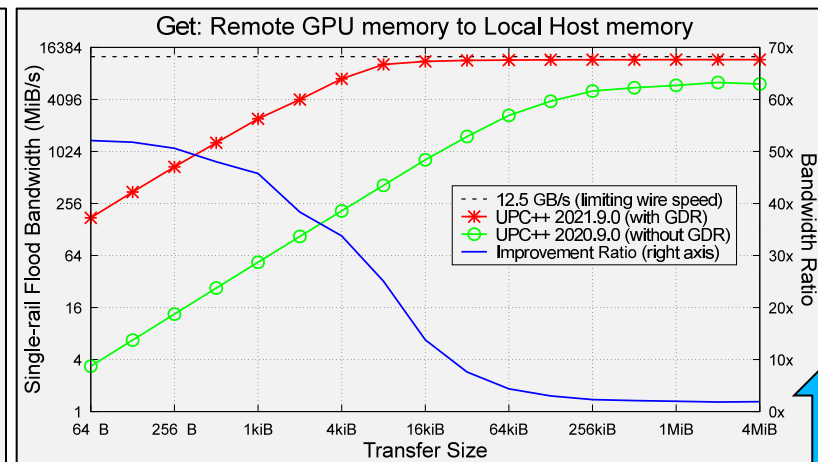
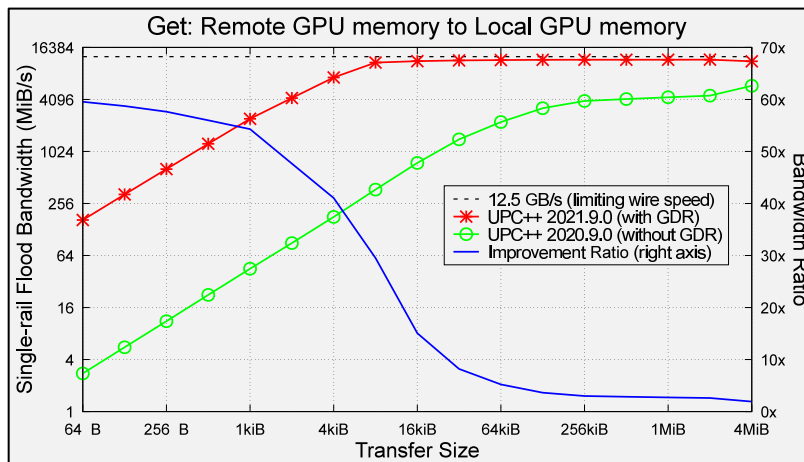
Static kind info in global_ptr type accelerates access

- Also have generic global_ptr<T, memory_kind::any> that can reference any kind of shared memory
- copy is polymorphic and can transfer data between memories anywhere in the system

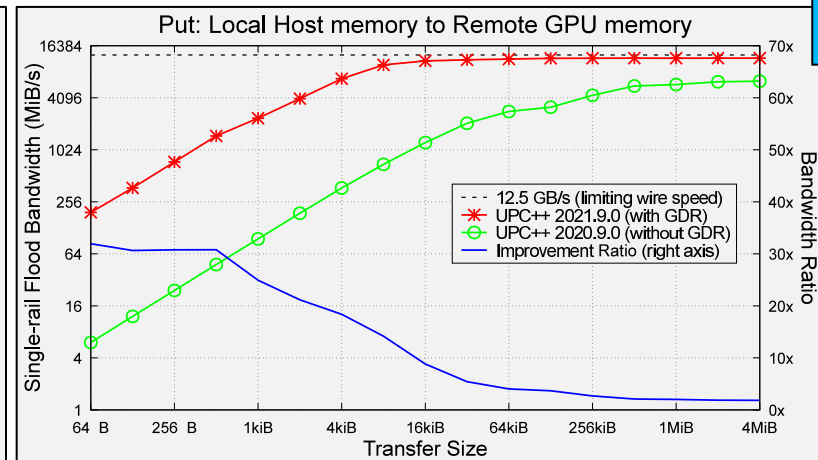
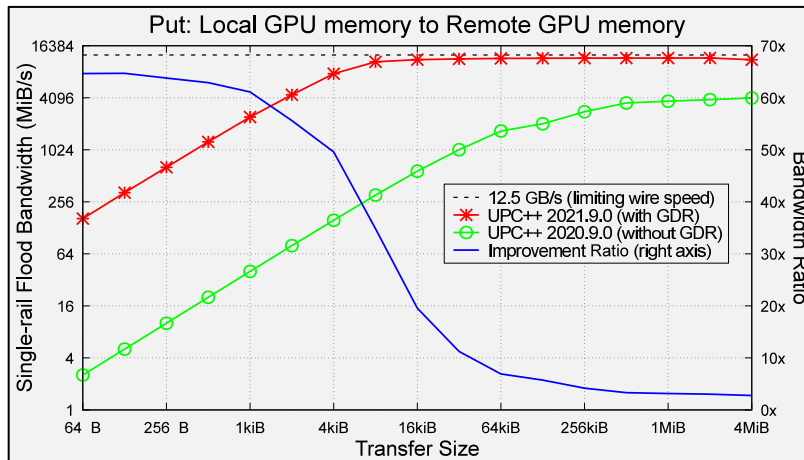
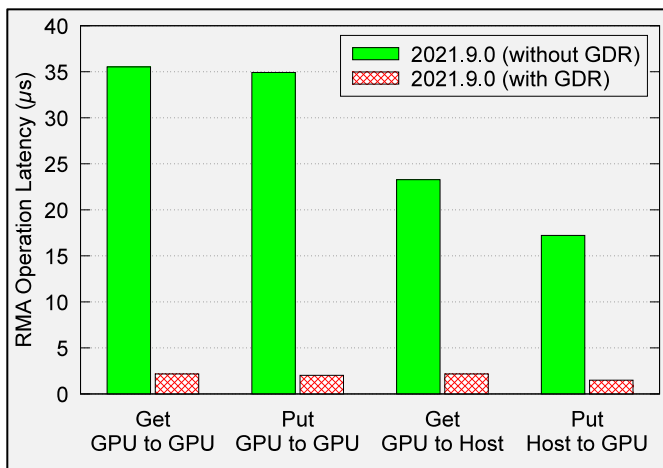
Microbenchmarks of upcxx: :copy on OLCF Summit

- UPC++ leverages GASNet-EX support for GPUDirect RDMA (GDR) to communicate to/from GPU memory without staging through host memory.
 - Large xfers: 2x better bandwidth
 - Small xfers: 11x to 17x better latency and up to 65x better bandwidth
- Get operations to/from GPU memory perform as well as for host memory.

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



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



All results were collecting using the version of the `cuda_benchmark` test that appears in the 2021.9.0 release. This test was run between two nodes of OLCF Summit, over its EDR InfiniBand network.

Paul H. Hargrove, Dan Bonachea, Colin A. MacLean, Daniel Waters. GASNet-EX Memory Kinds: Support for Device Memory in PGAS Programming Models, Poster to appear at SC21, Nov 2021. <https://doi.org/10.25344/S4P306>

Application Kernel Study using `upcxx::copy`

Study converted a Kokkos tutorial example from use of MPI message passing to UPC++ RMA to demonstrate interoperability with Kokkos

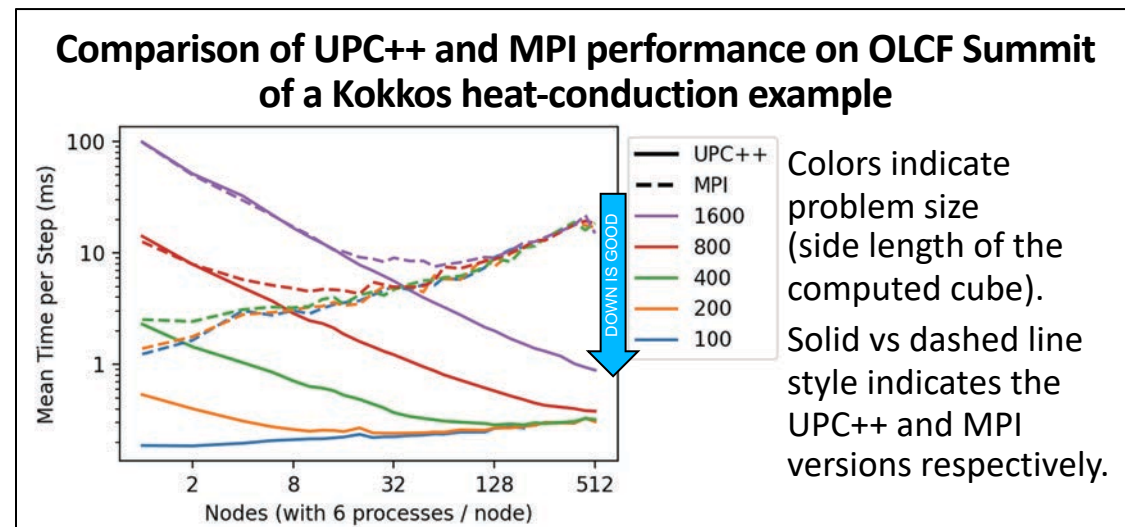
- Use of UPC++ memory kinds for device memory management
- Use of `upcxx::copy` for one-sided data movement and remote notification

Despite no changes to computation, saw unexpected performance differences

- Better strong scaling (see figure)
- Much lower variance in performance

More complete information to appear in a workshop talk on Friday morning:

D. Waters, C. A. MacLean, D. Bonachea, P. H. Hargrove. **"Demonstrating UPC++/Kokkos Interoperability in a Heat Conduction Simulation (Extended Abstract)"**, In *2021 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI+X (PAW-ATM)*, St. Louis, MO, Nov 2021. <https://doi.org/10.25344/S4630V>



UPC++ additional resources

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

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

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

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

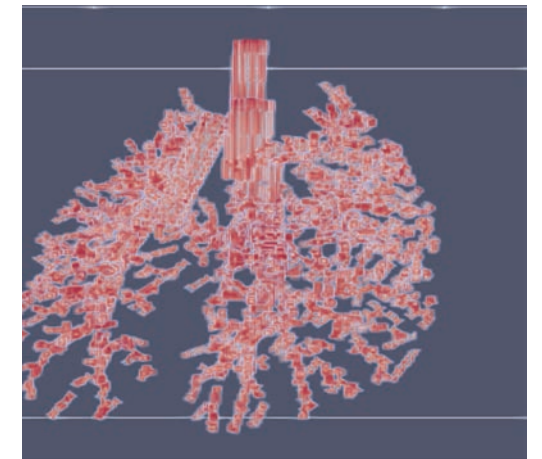
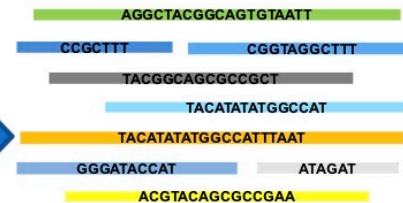
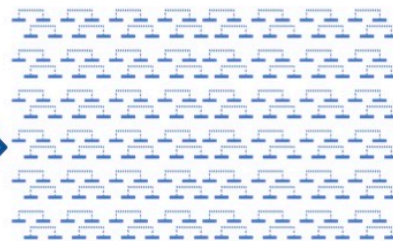
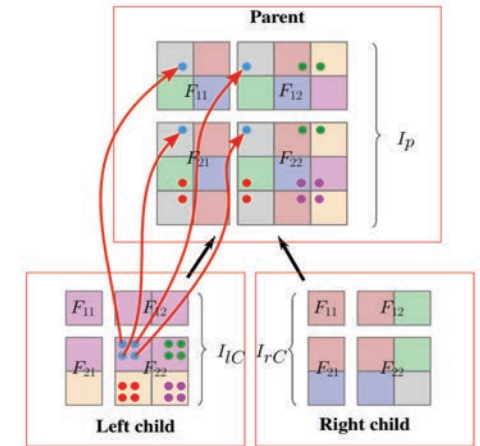
Application Case Studies

Application case studies

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

We discuss several applications written in UPC++:

- symPack, a sparse symmetric matrix solver
- SIMCoV, agent-based simulation of lungs with COVID
- MetaHipMer, a genome assembler



Sparse multifrontal direct linear solver

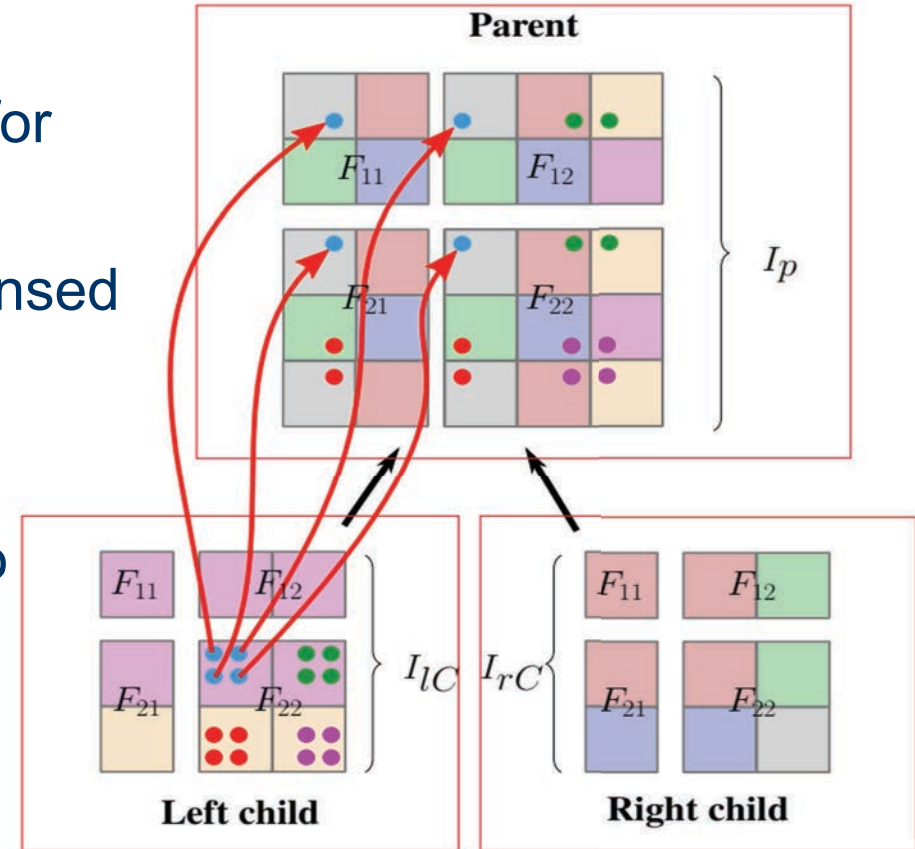
Sparse matrix factorizations have low computational intensity and irregular communication patterns

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>

Implementation of the extend-add operation

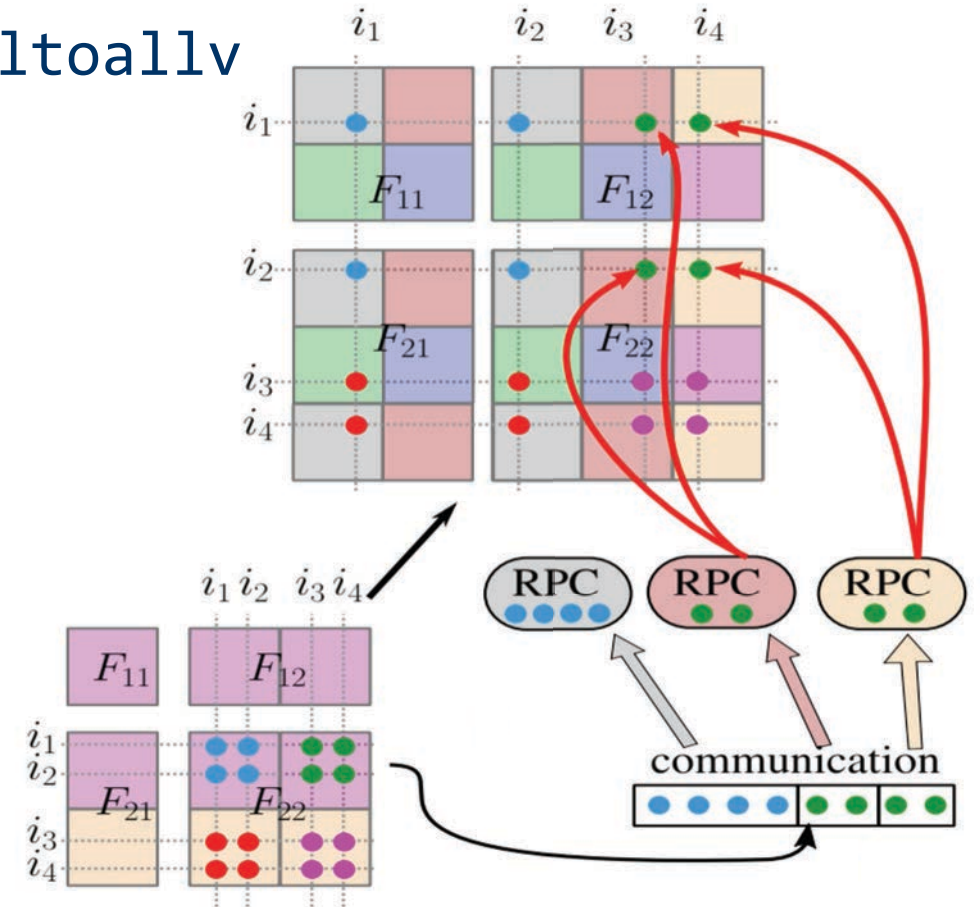
Data is binned into per-destination contiguous buffers

Traditional MPI implementation uses `MPI_Alltoallv`

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

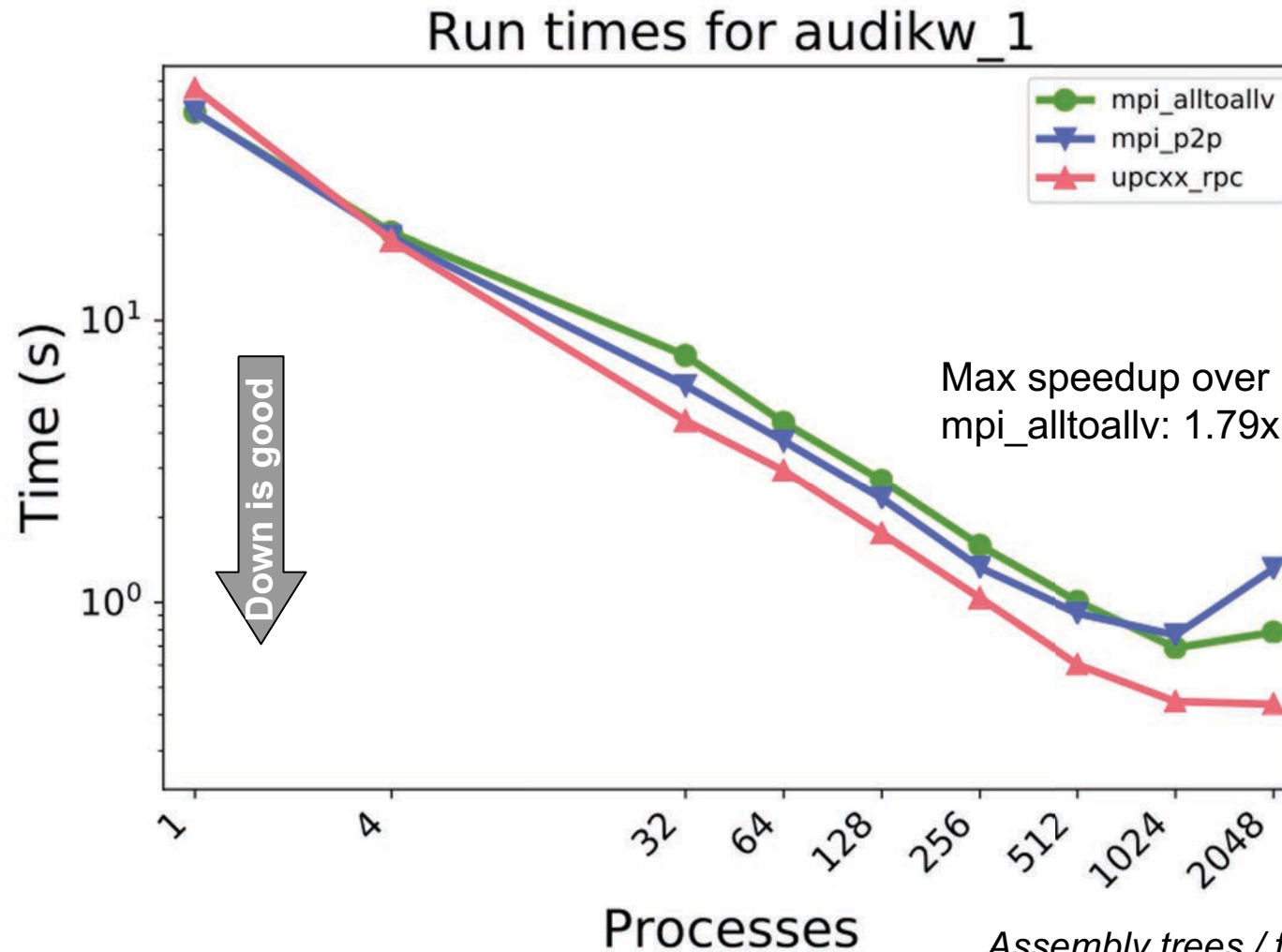
UPC++ Implementation:

- 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 (extend-add)

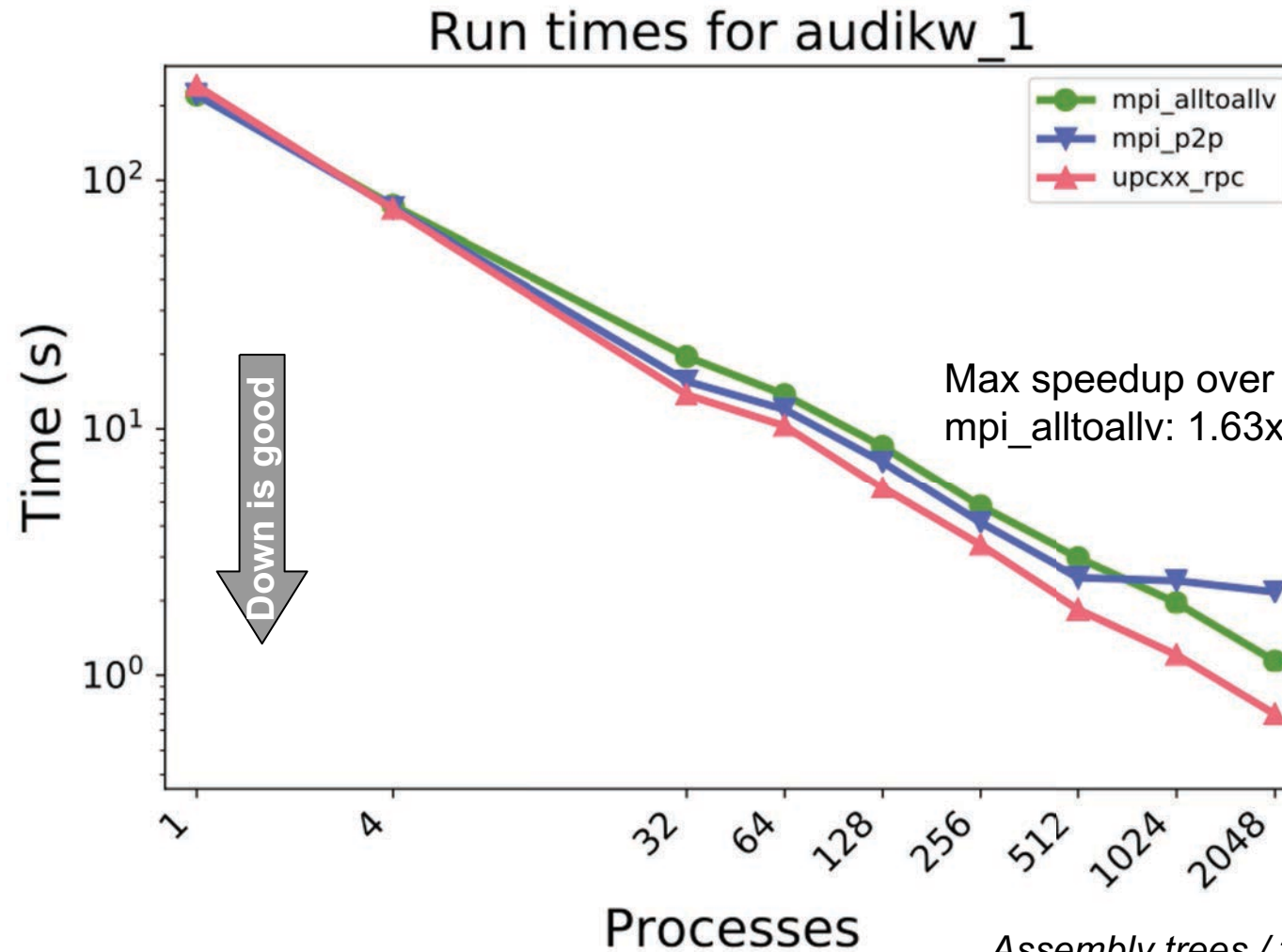


Experiment done on
NERSC Cori Haswell
Cray XC Aries

Assembly trees / frontal matrices
extracted from STRUMPACK

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

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



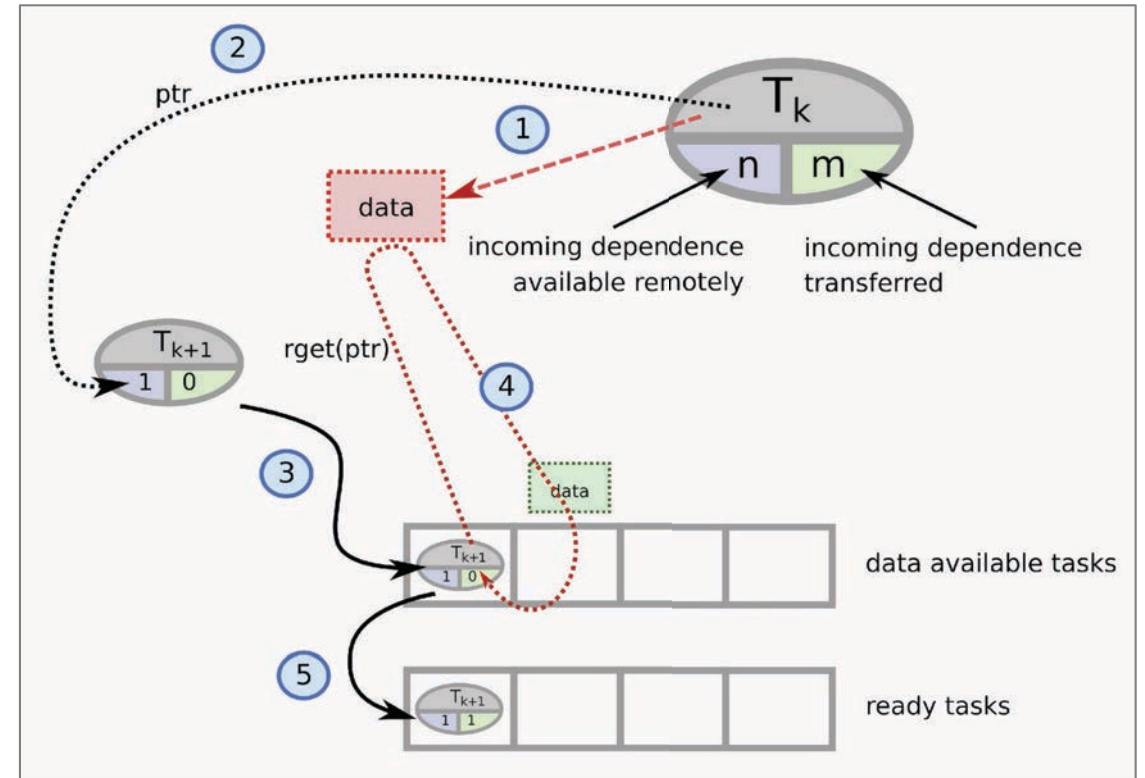
Experiment done on
NERSC Cori KNL
Cray XC Aries

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 to 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

Yelick, Kamil, Rouson / UPC++ / SC21 Tutorial / upcxx.lbl.gov

symPACK a solver for sparse symmetric matrices

Matrix is distributed by supernodes

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

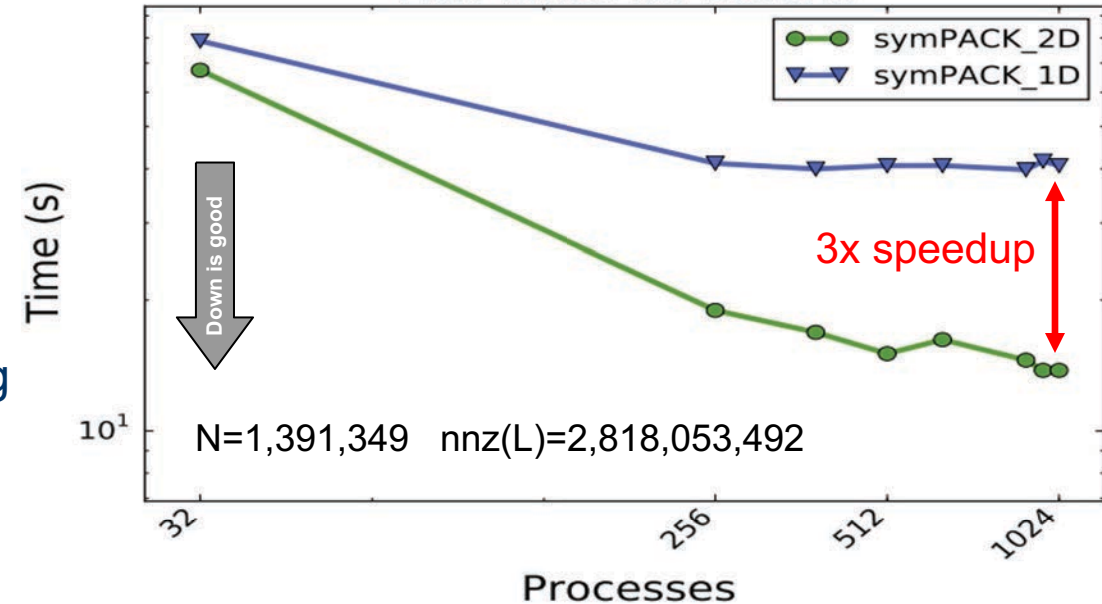
Strong scalability on Cori Haswell:

- Up to 3x speedup for Serena
- Up to 2.5x speedup for DG_Phosphorene_14000

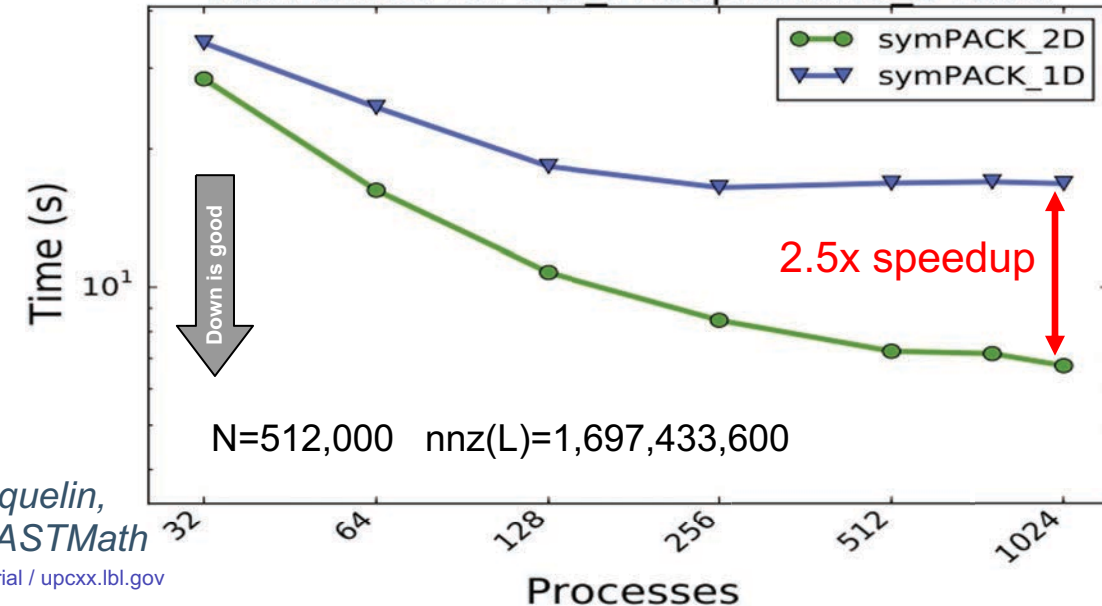
UPC++ enables the finer granularity task graph to be fully exploited

- **Better strong scalability**

Run times for Serena



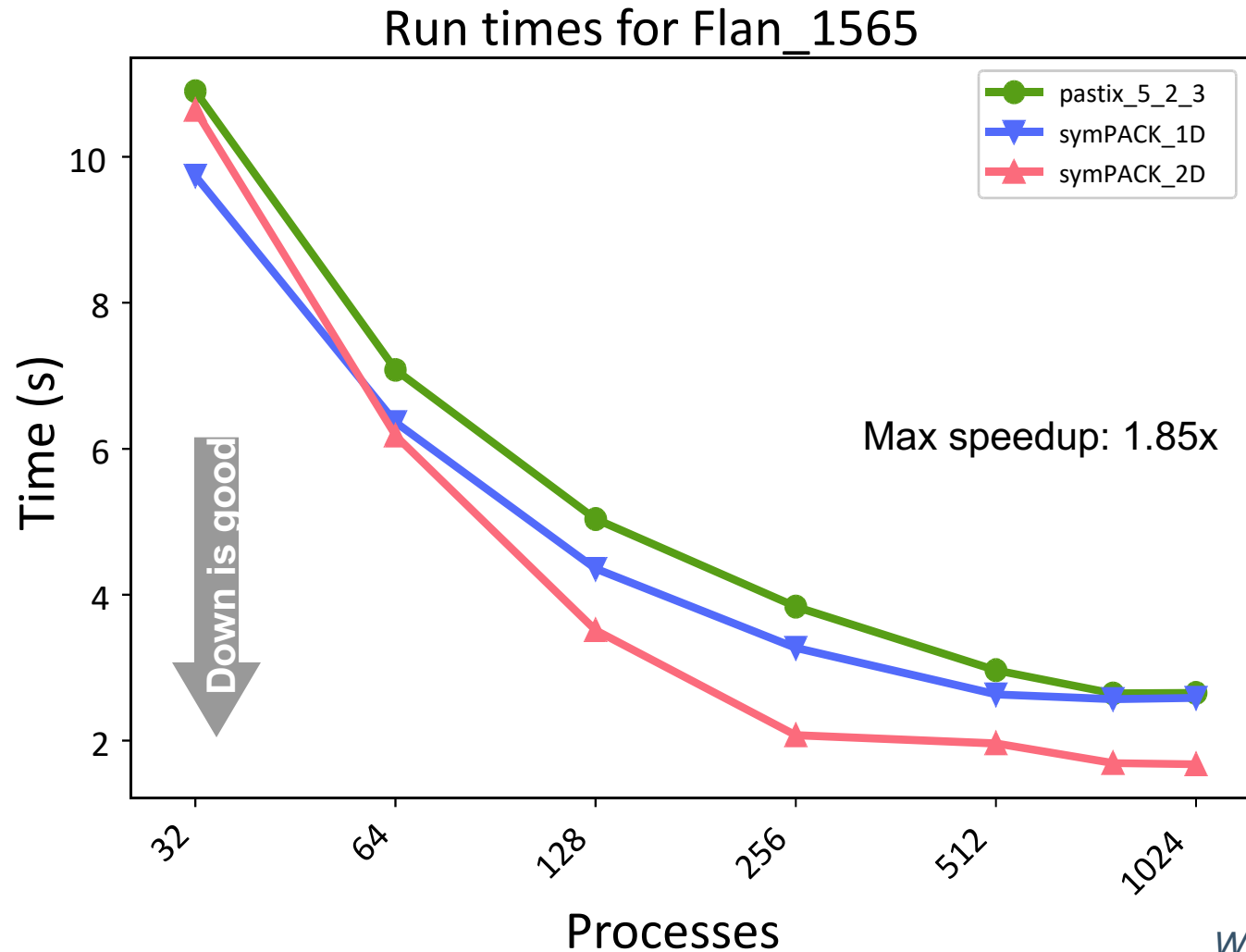
Run times for DG_Phosphorene_14000



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

Yelick, Kamil, Rouson / UPC++ / SC21 Tutorial / upcxx.lbl.gov

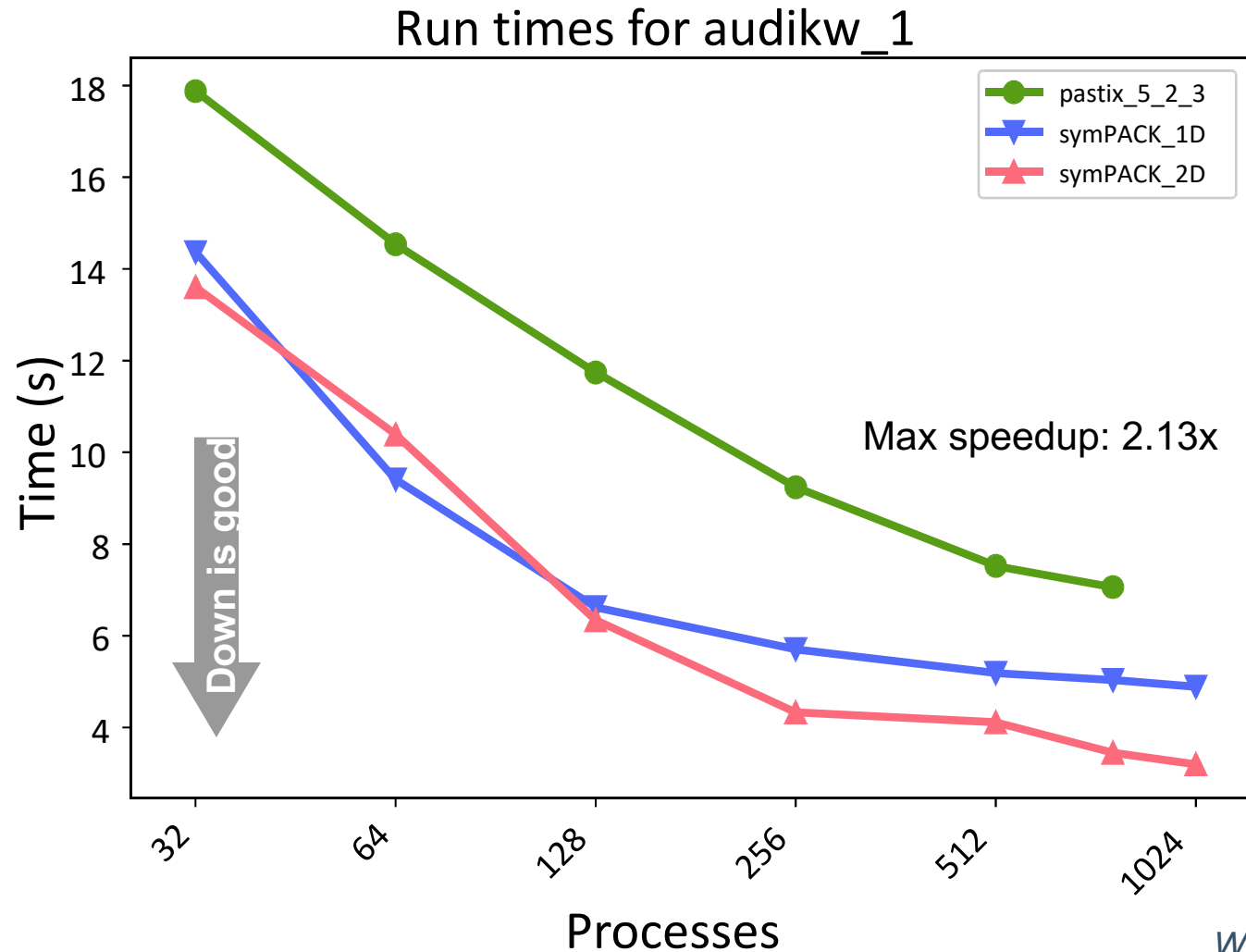
symPACK strong scaling experiment



Experiment done on
NERSC Cori KNL
Cray XC Aries

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

symPACK strong scaling experiment



Experiment done on
NERSC Cori Haswell
Cray XC Aries

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

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
- symPACK outperforms other sparse symmetric solvers

<https://upcxx.lbl.gov/sympack>

SIMCoV: Spatial Model of Immune Response to Viral Lung Infection

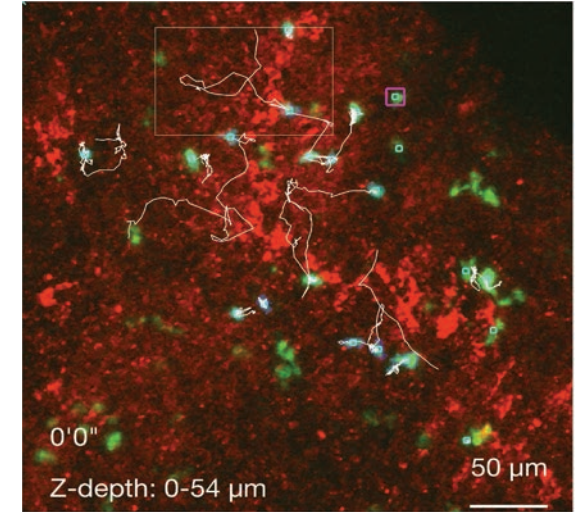
M. Moses, J. Cannon (UNM), S. Forrest (ASU) and S. Hofmeyr (LBNL)

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

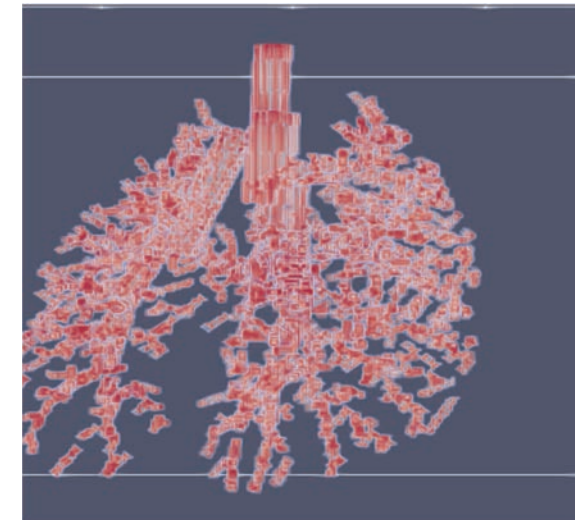
To investigate this, we have built a 3D agent-based model of the lungs, called SIMCoV and available at <https://github.com/AdaptiveComputationLab/simcov>

SIMCoV 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 minutes for 20 to 30 days
- SIMCoV in UPC++
 - Distributed 3D spatial grid
 - Particles move over time, but computation is localized
 - Load balancing is tricky: active near infections
- UPC++ benefits:
 - Heavily uses RPCs
 - Easy to develop first prototype
 - Good distributed performance and avoids explicit locking
 - Extensive support for asynchrony improves computation/communication overlap

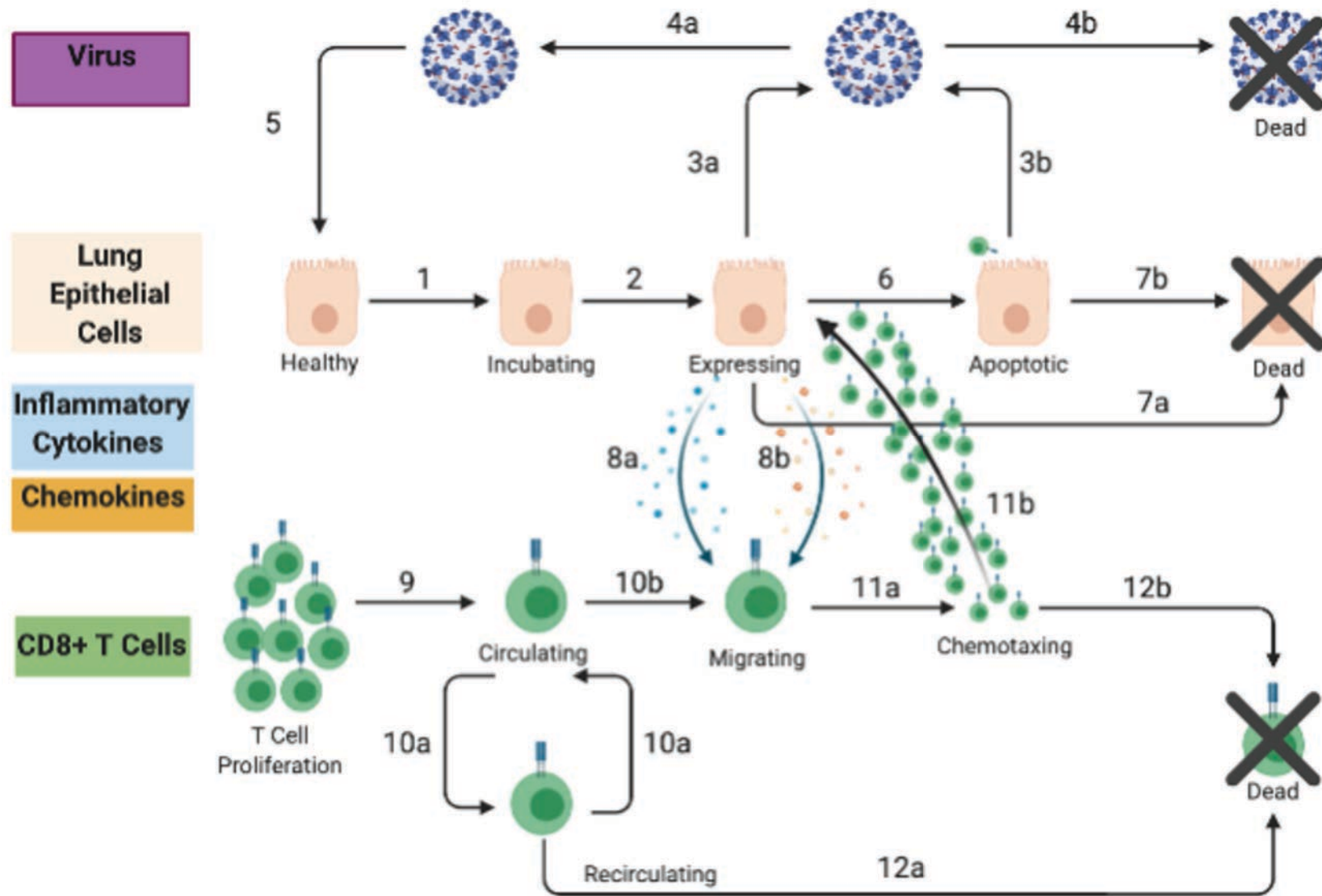


Imaging of T cell movement in lung tissue



Fractal model of airways in lung

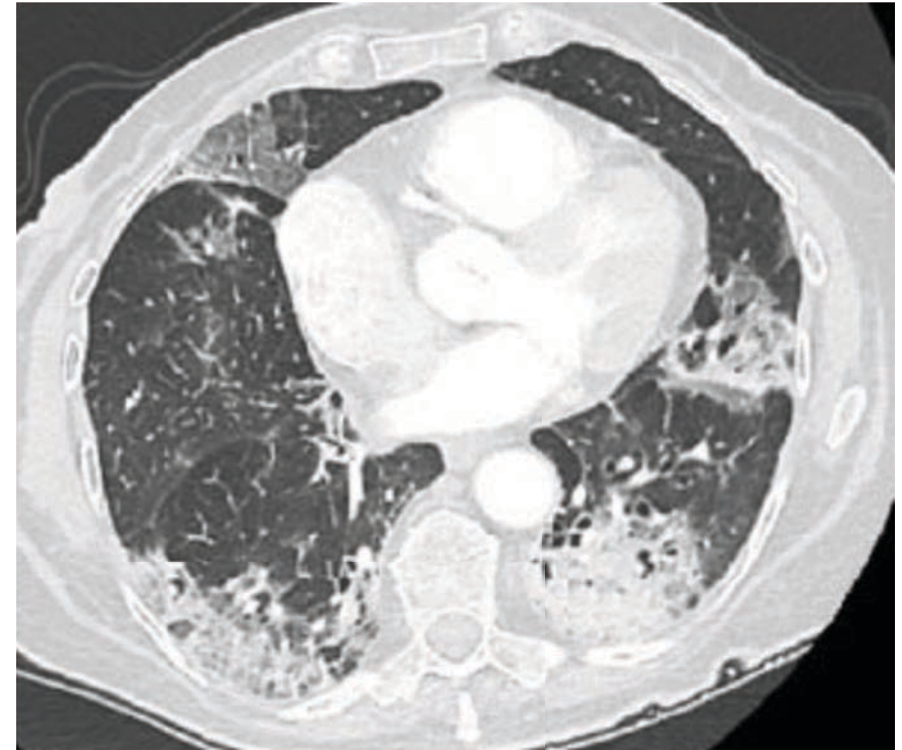
SIMCoV Components



Use of Observational Data

Observational data is used in three ways:

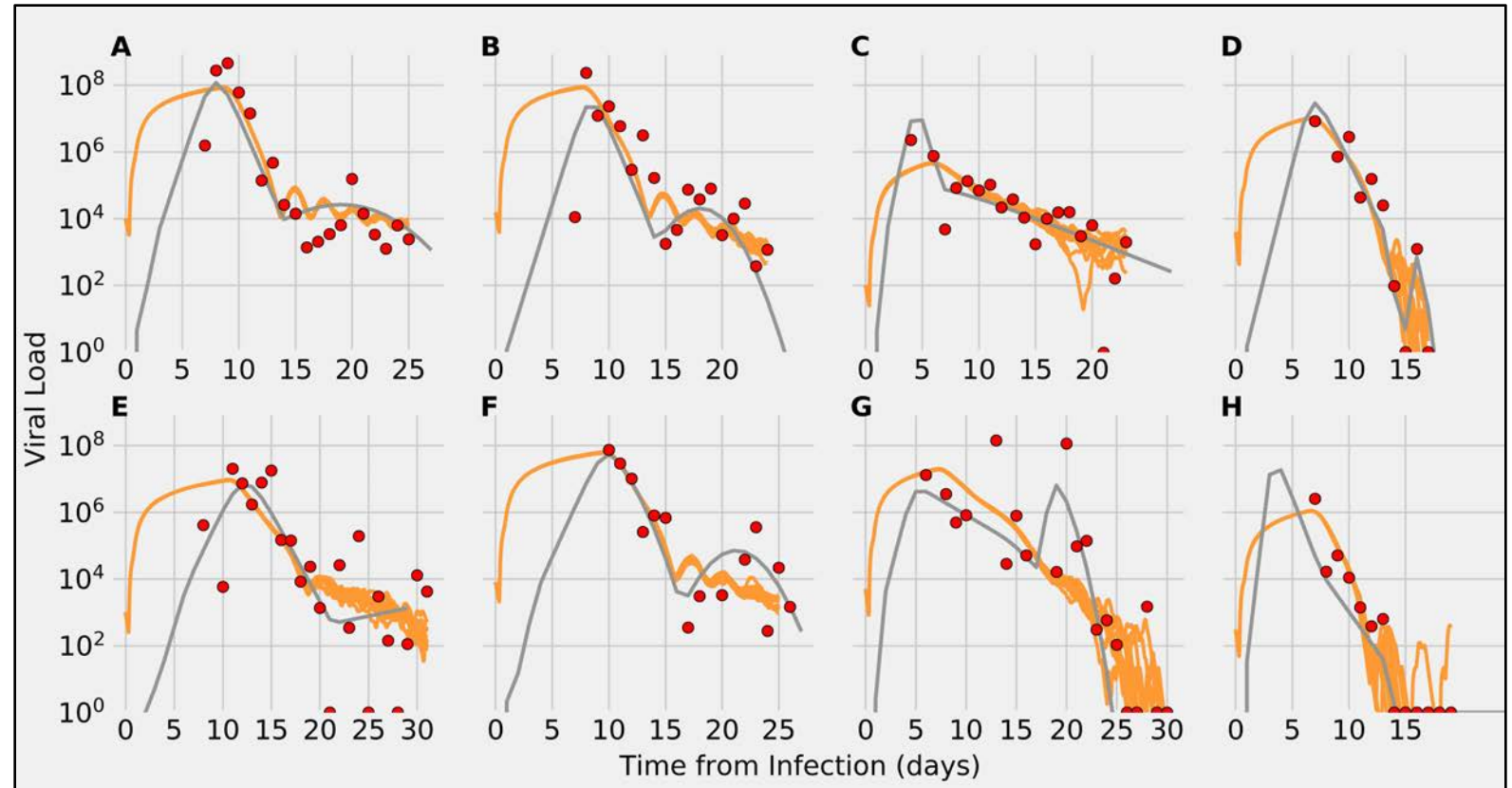
- To obtain parameters for the model
 - e.g. rate of viral production by infected cells, T cell generation rate, rate of T cell movement, etc.
- To validate the model
 - does the output "look" like a typical Covid-19 infection? e.g. distribution of plaques
 - are the measured quantities similar with similar time courses? e.g. viral load
- To seed the model
 - Given an initial distribution of the virus:
 - what is the most likely outcome?
 - what is the best intervention strategy?



Lung CT showing sites of infection

Viral load predictions from SIMCoV simulations

- Using just four model parameters, SIMCoV simulations (orange curves) reproduce viral load observations (red dots) in eight patients
- Goodness-of-fit metrics similar to those of prior models (grey curves)
- Earlier rise to peak levels consistent with observation that infection is most contagious in early stages, before onset of symptoms



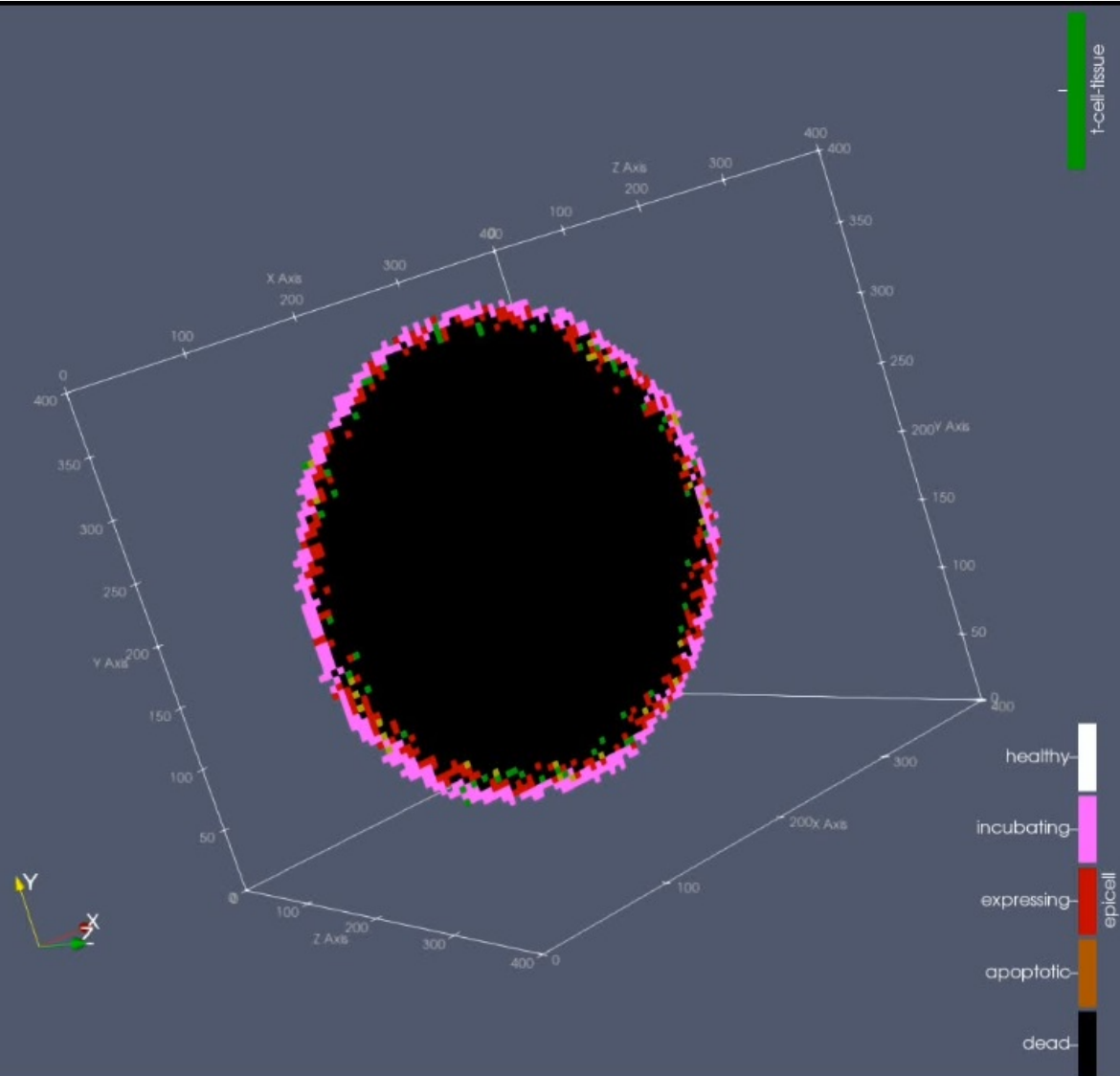
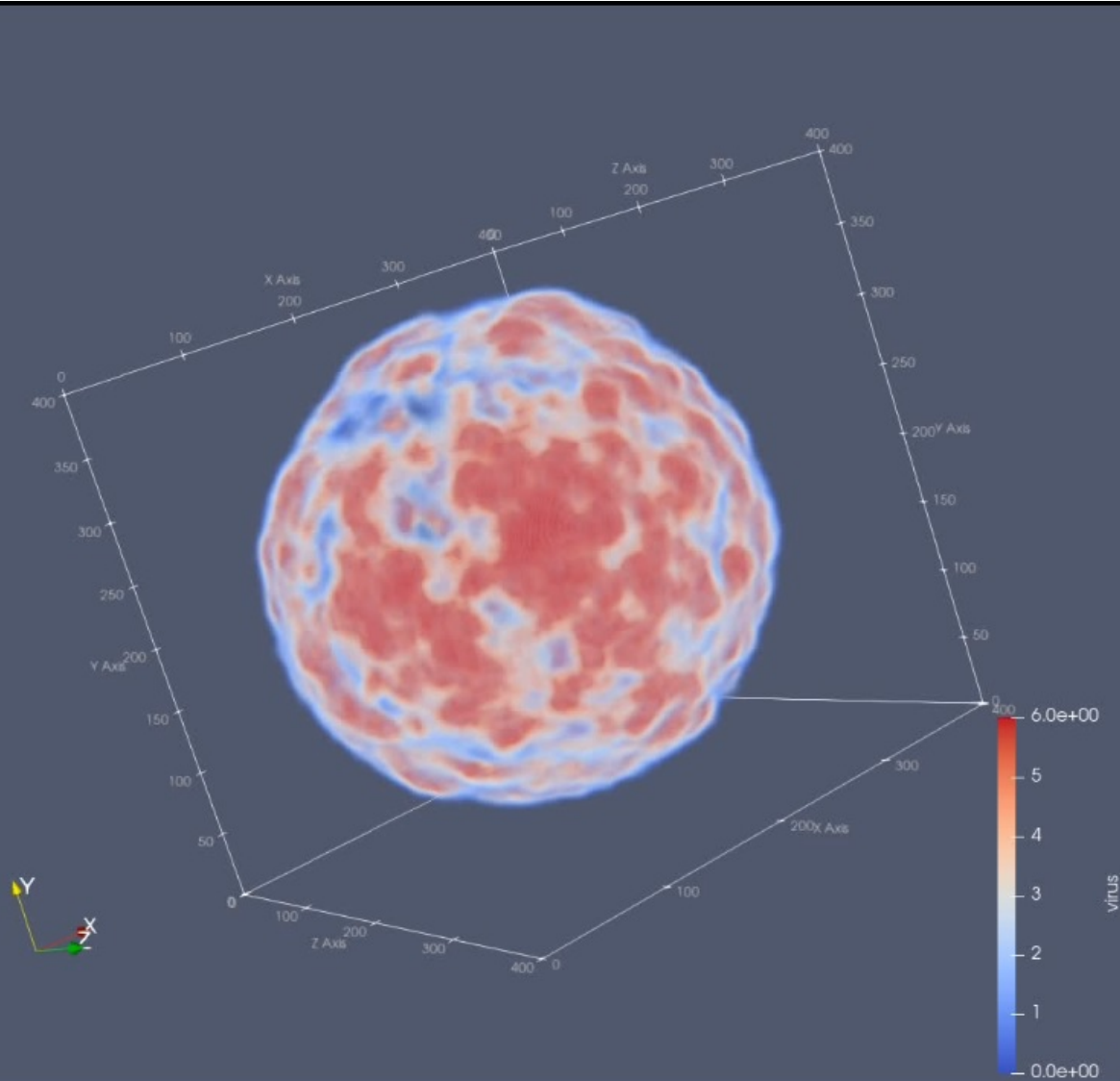
M. E. Moses, S. Hofmeyr, J. L. Cannon, A. Andrews, R. Gridley, M. Hinga, K. Leyba, A. Pribisova, V. Surjadidjaja, H. Tasnim, S. Forrest.
"Spatially distributed infection increases viral load in a computational model of SARS-CoV-2 lung infection"
bioRxiv 2021.05.19.444569; doi: <https://doi.org/10.1101/2021.05.19.444569>

Visualization of Prototype Simulation

Run headless and visualize afterwards using Paraview

Spread of infection from single focal point

Very small 2D area without branching structures



ExaBiome: Exascale Solutions for Microbiome Analysis



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



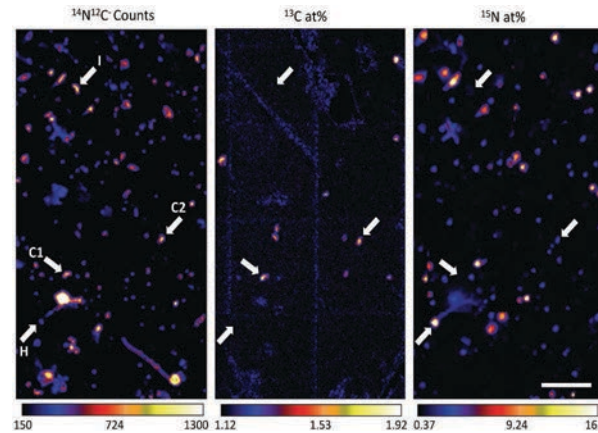
How do microbes affect disease and growth of switchgrass for biofuels? (4TB)



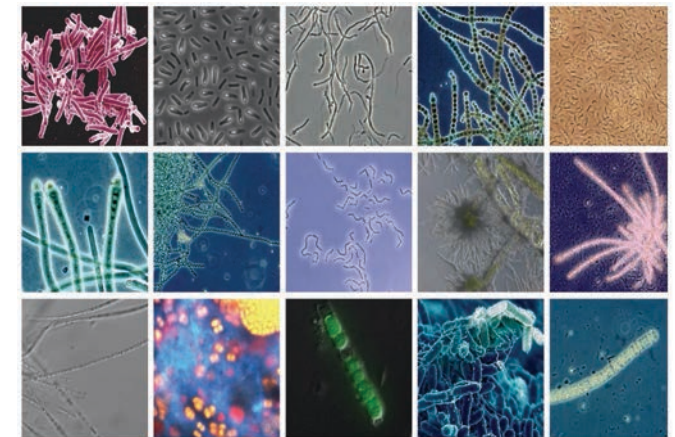
How do microbial dynamics change in freshwater lakes across 17 years? (25TB)



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



Combine genomics with isotope tracing methods for improved functional understanding (16TB)



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

De Novo genome assembly problem

Input

GCTACGGAATAAAACCAGGAACAACAGAGCC_AGCAC

ATAAAACCAGGTACAACAGACCCAGCACGGATCCA

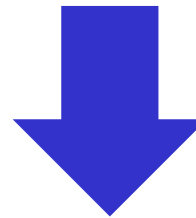
GC_ACGGAATACAACCAGGAACAACAGACCCAGCAC

GAACAACAGACCCAGCATGGATCCA



reads
(input, typically
100-250 chars)

*Multiple
copies
(20x typical)*



GCTACGGAATAAAACCAGGAACAACAGACCCAGCACGGATCCA

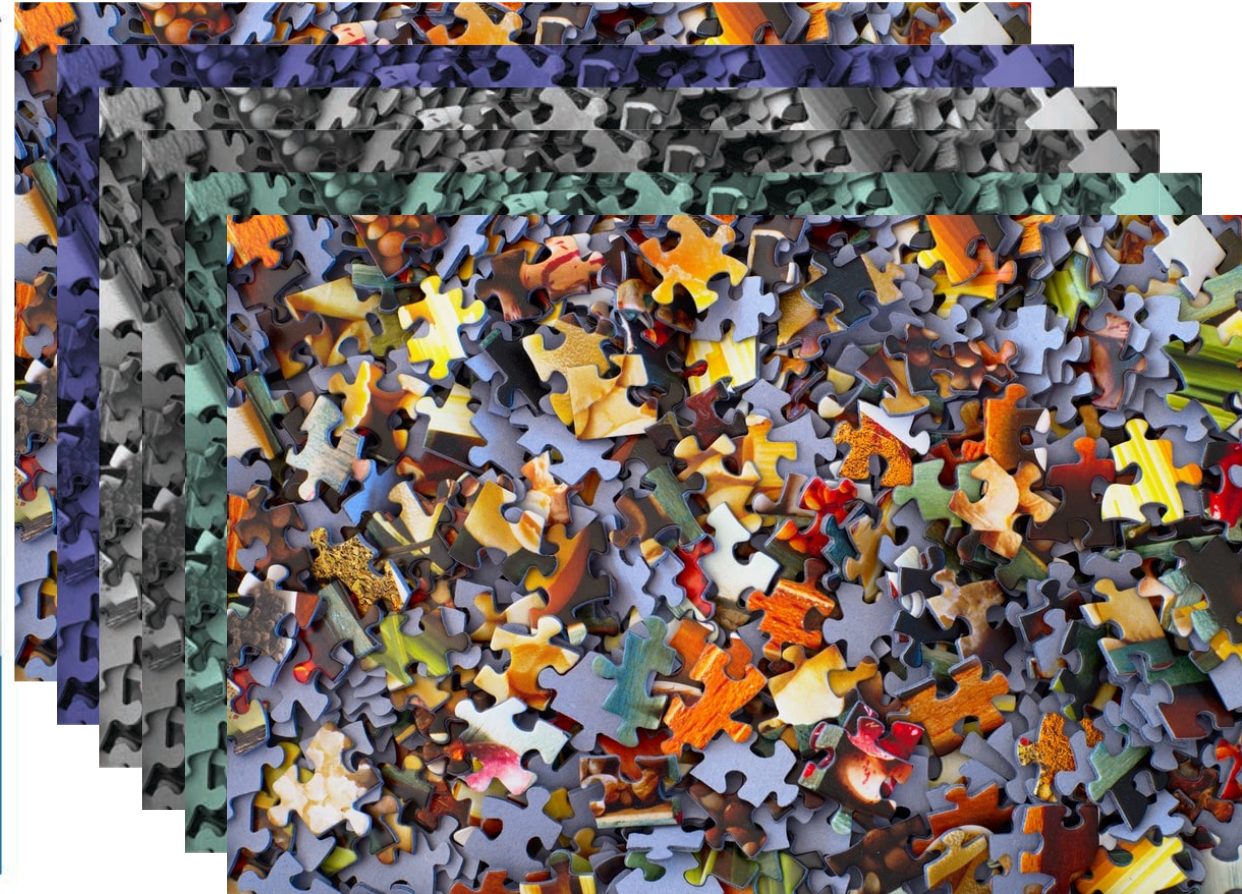
Output

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

De Novo Genome Assembly



De Novo **Meta**Genome Assembly



MetaHipMer: The First HPC MetaGenome Assembler



Multiassembly: 1 piece at a time on an SMP

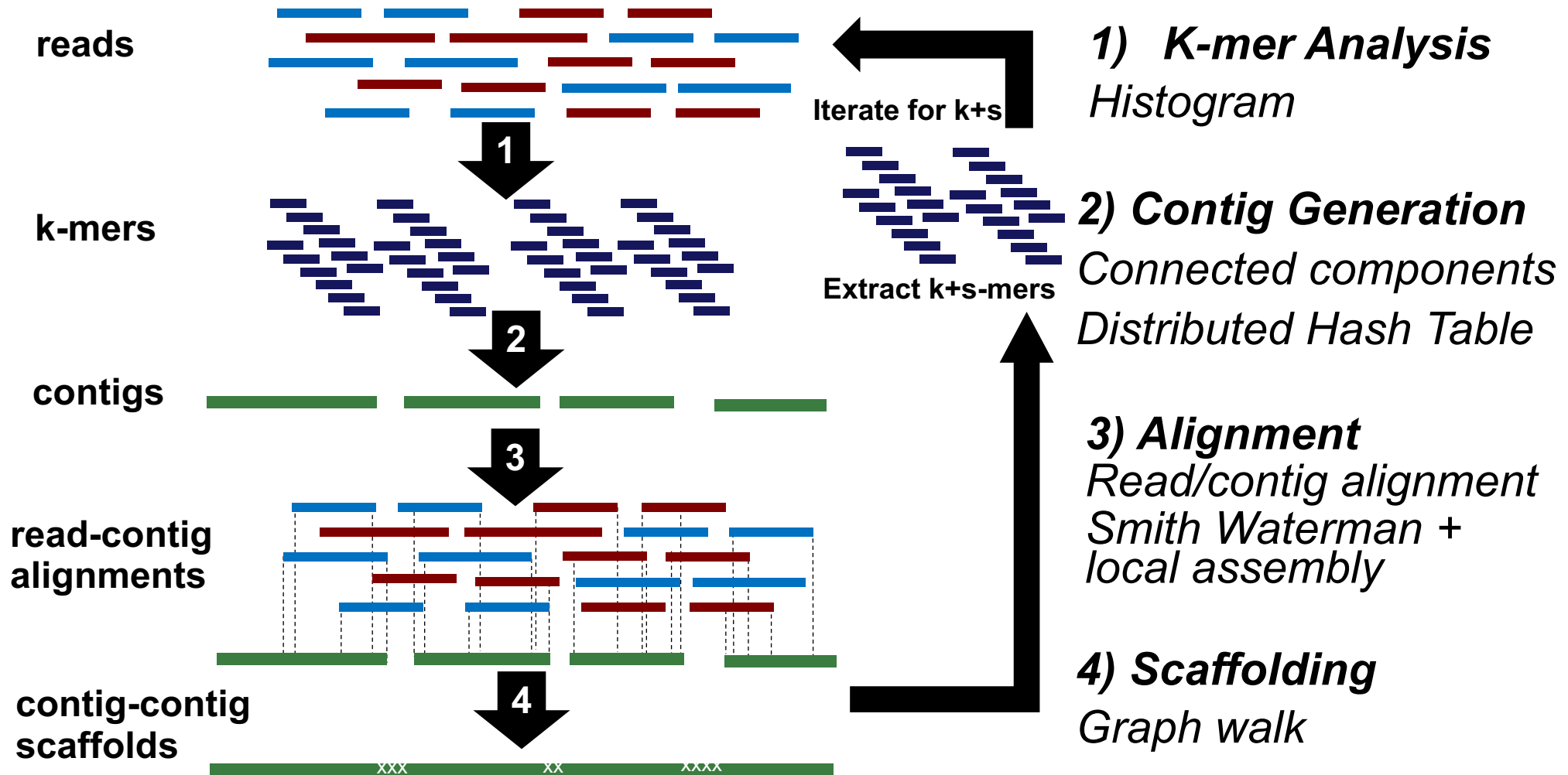


Coassembly all assembled together on HPC

MetaHipMer: Terabyte data sets + HPC systems + scalable algorithms = better quality results

SC18 and SC21 best paper finalists!

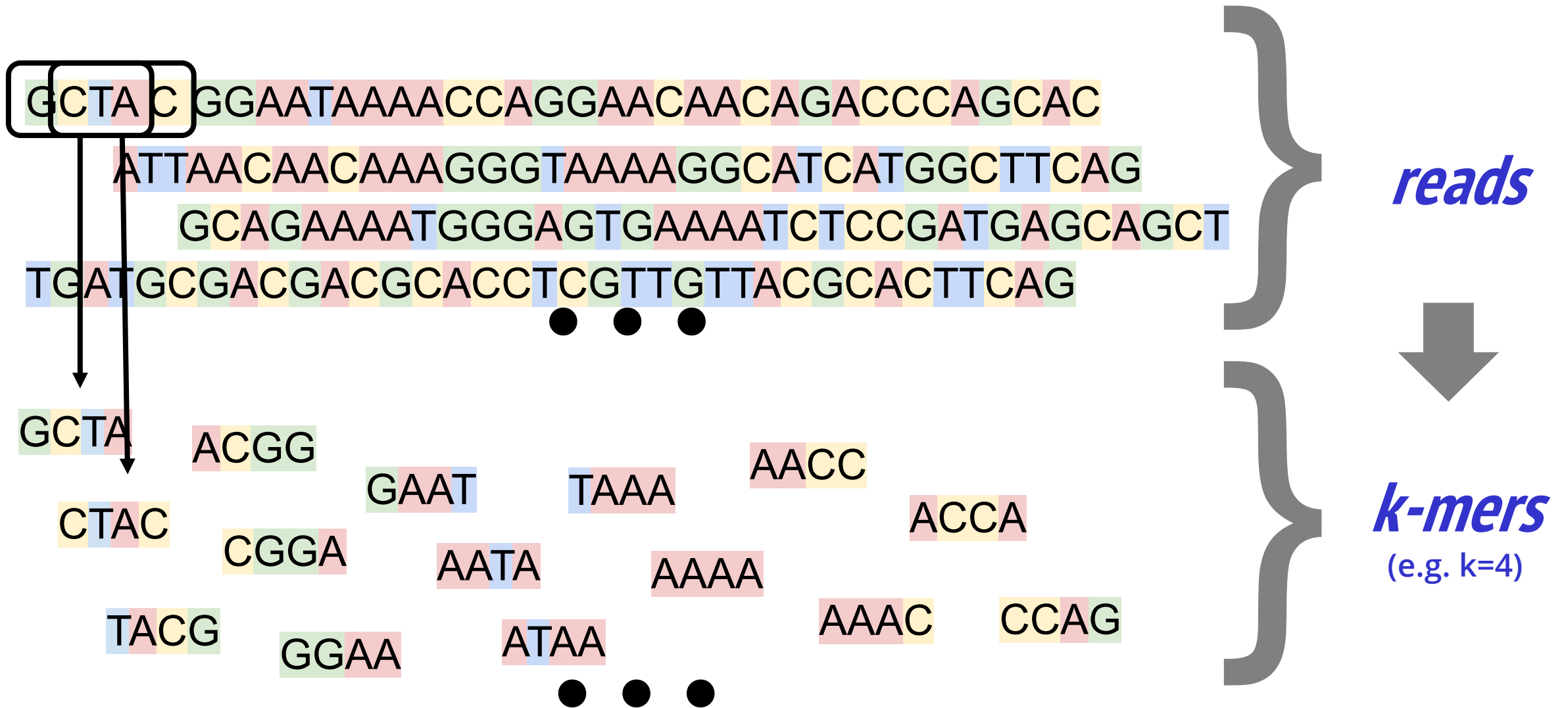
MetaHipMer MetaGenome Assembler



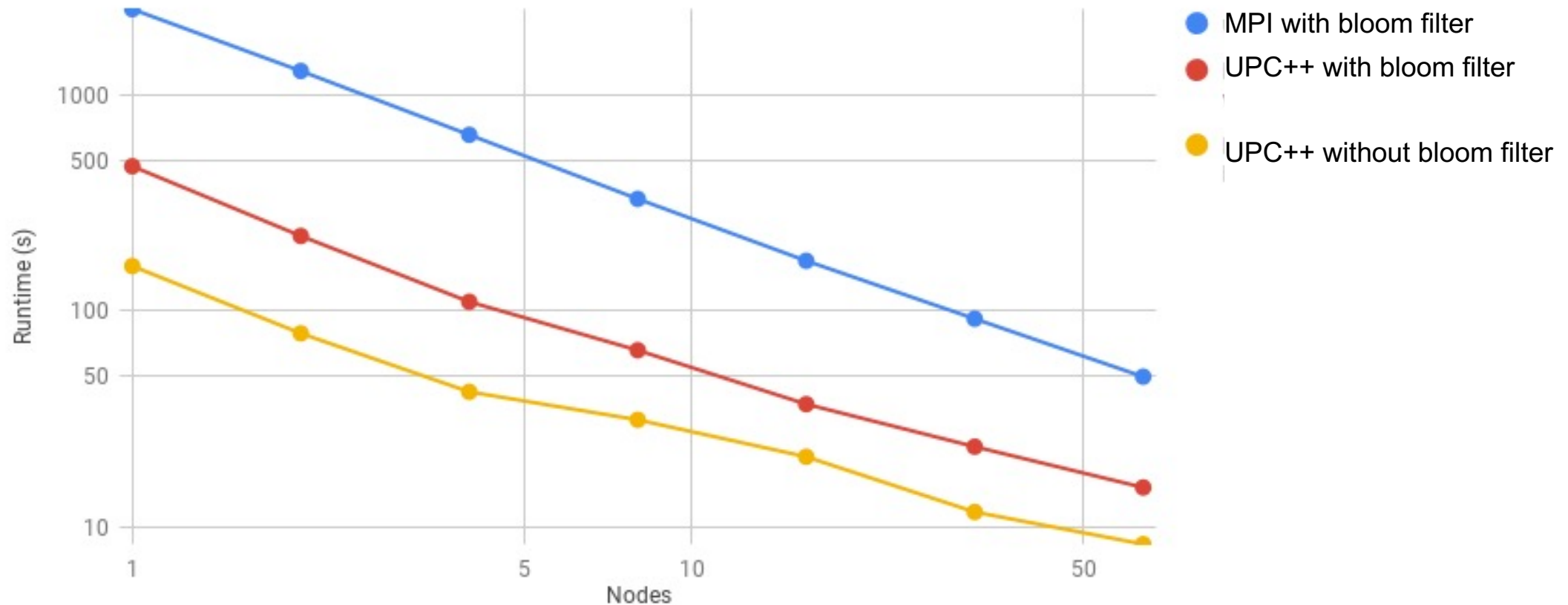
Originally written in MPI & UPC, now in UPC++

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

K-Mer Analysis Uses a Distributed Hash Table



K-mer counting in UPC++

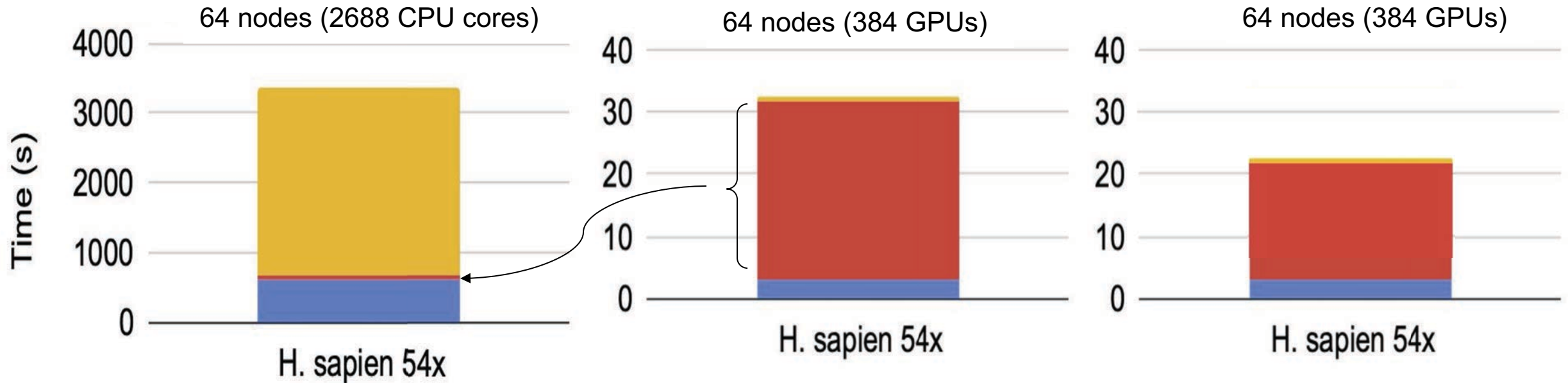


- Compared against bulk-synchronous version in MPI
- The UPC++ version avoids barriers, and saves memory (no MPI runtime)
- It's faster and simpler!

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

K-mer Counting: GPU Parallelism

■ parse & process kmers ■ kmer counter ■ exchange



- 100x speedup from GPU optimizations; becomes communication bound
- Reduce communication cost with locality-aware hashing (minimizers) and sending multiple pre-parsed (packed) k-mers (called supermers)

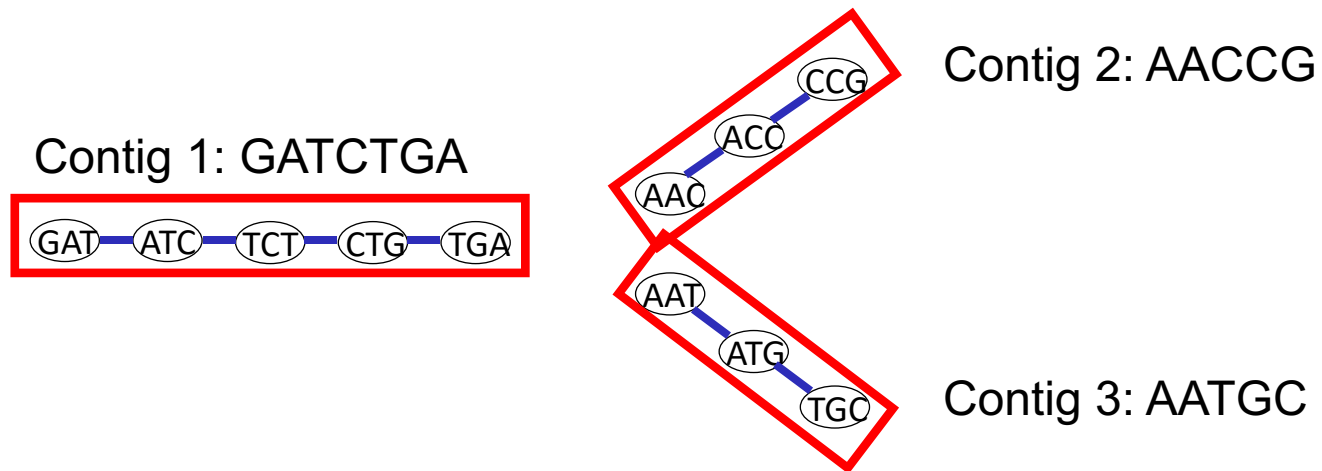
150x speedup!!

Distributed De Bruijn Graph

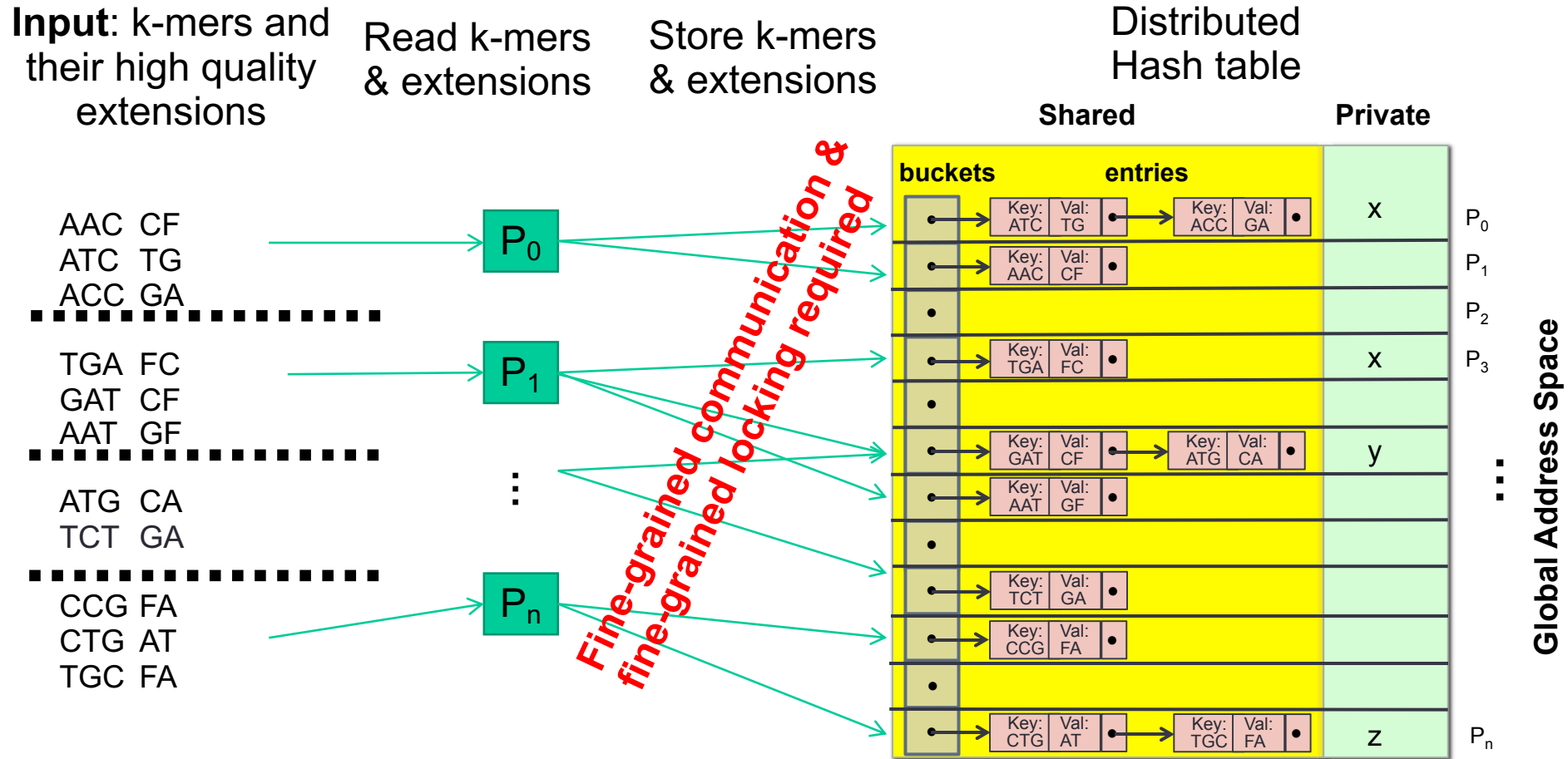
The **de Bruijn graph** of k-mers is represented as a hash table

- A k-mer is a node in a graph \Leftrightarrow a k-mer is an entry (key) in the hash table
- It stores the left and right “extension” (ACTG) as the value in the table

The connected components represent **contigs**.



Parallel De Bruijn Graph Construction



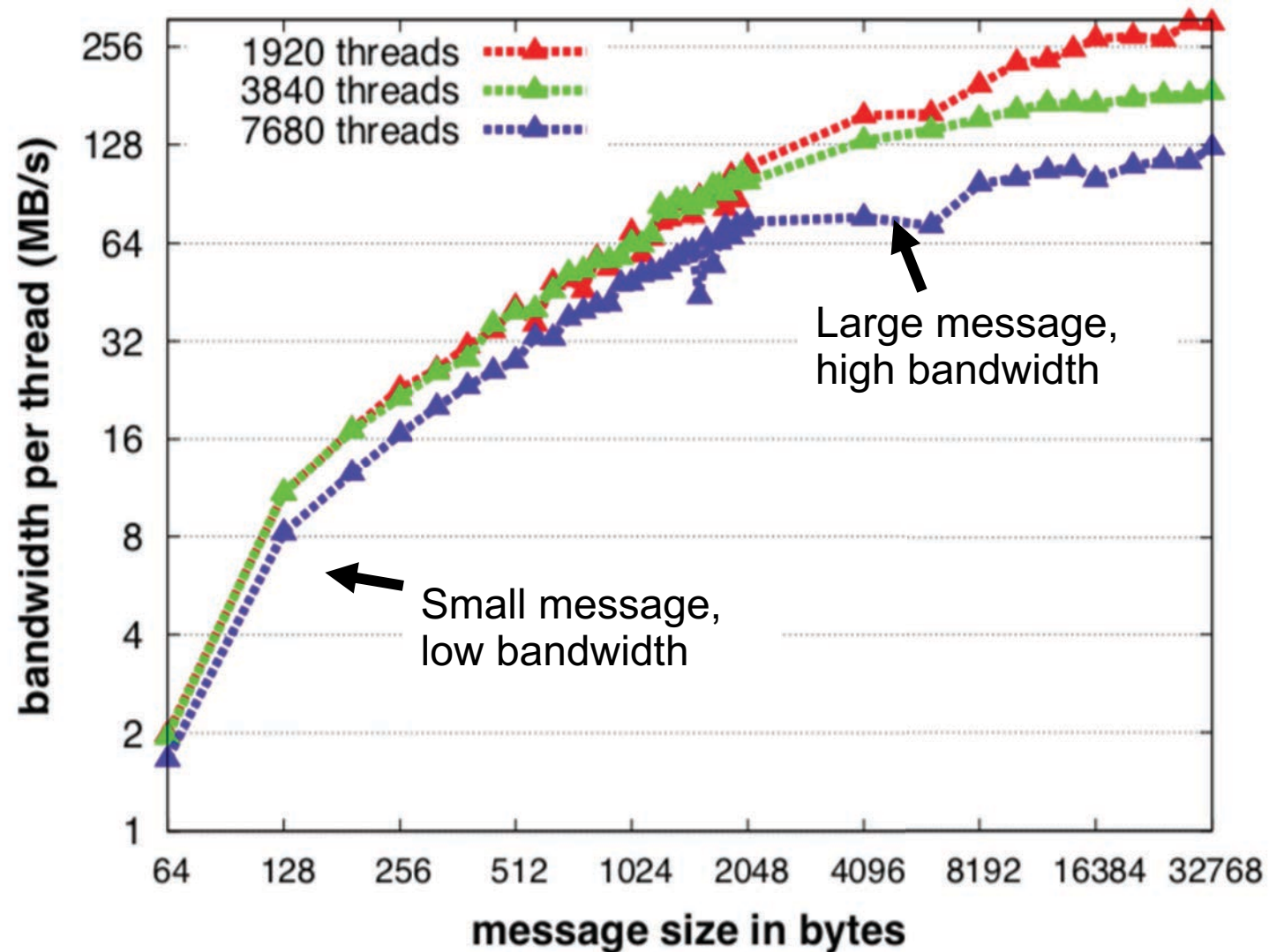
Distributed Hashmap in MetaHipMer

Memory-limited graph stages

- k-mers, contig, scaffolding

Optimized graph construction

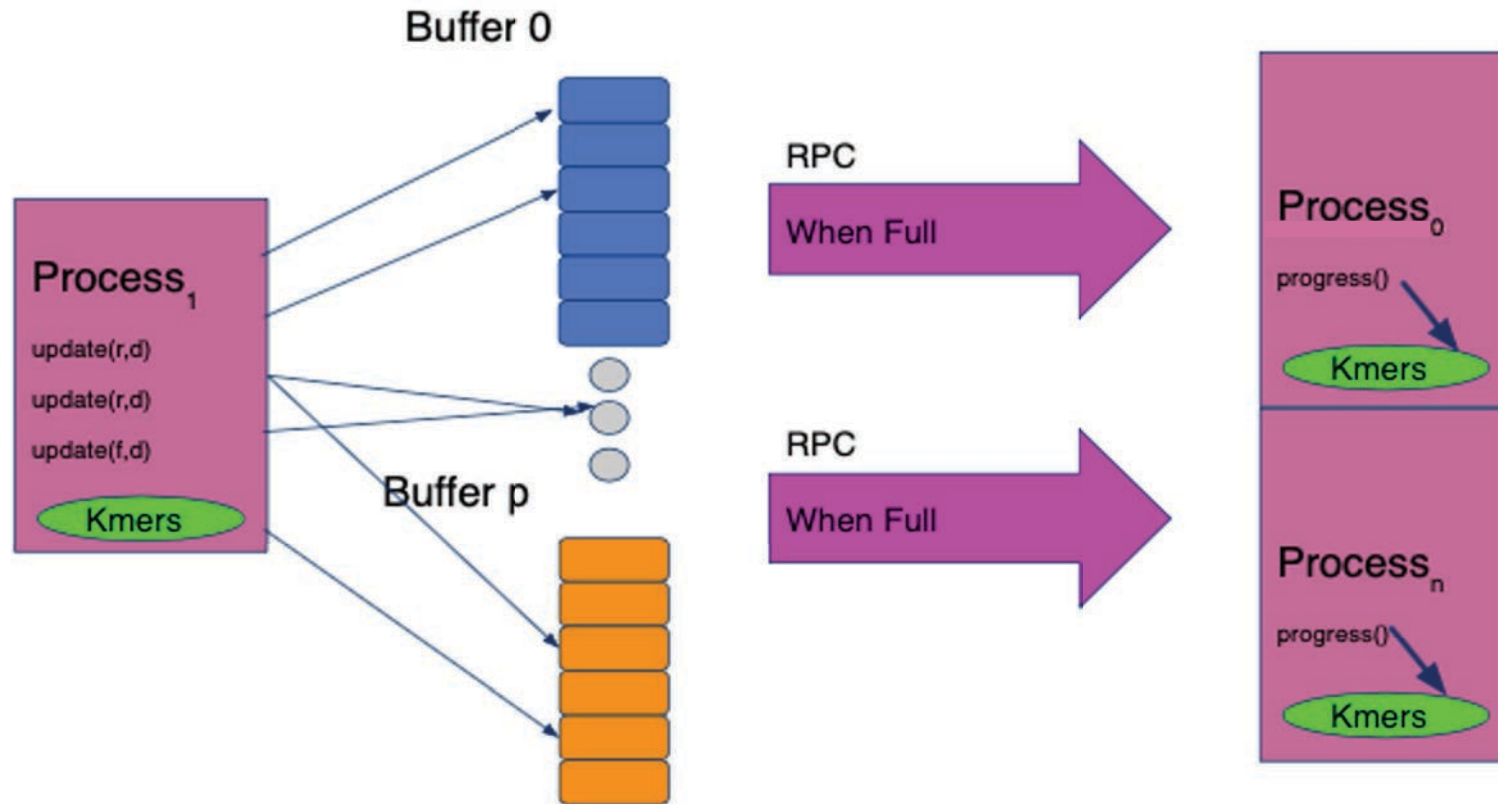
- Larger messages for better network bandwidth



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);

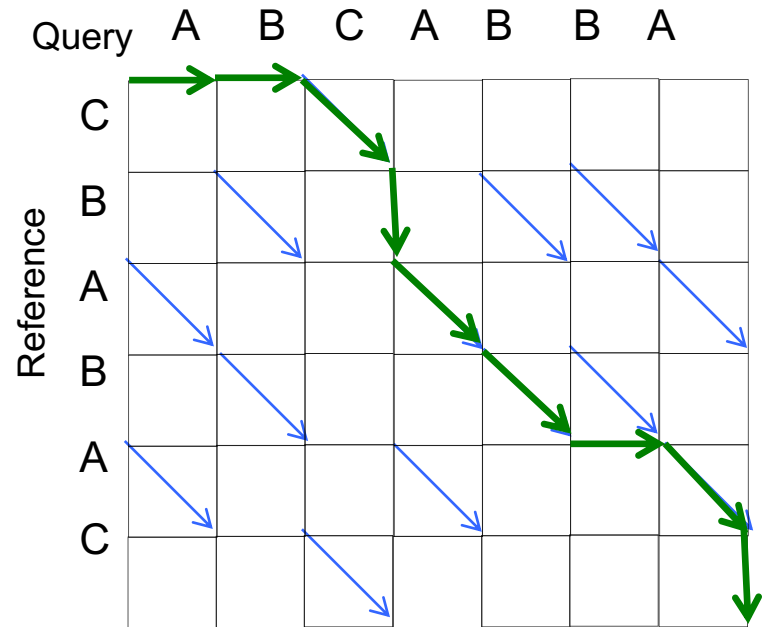
// add one element to the AggrStore
void update(intrank_t target_rank, T &elem);

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

Distributed Alignment: Hash Tables and Alignment

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

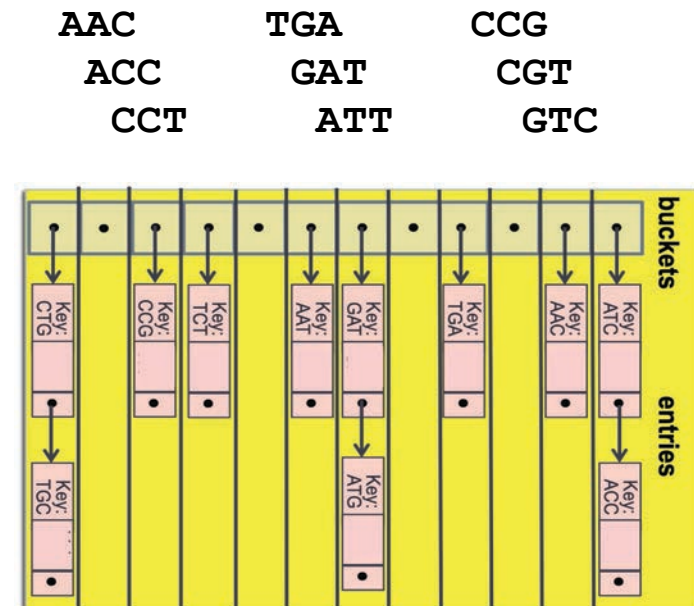
Dynamic programming on short strings with early stopping for bad alignments



Many variations of both!

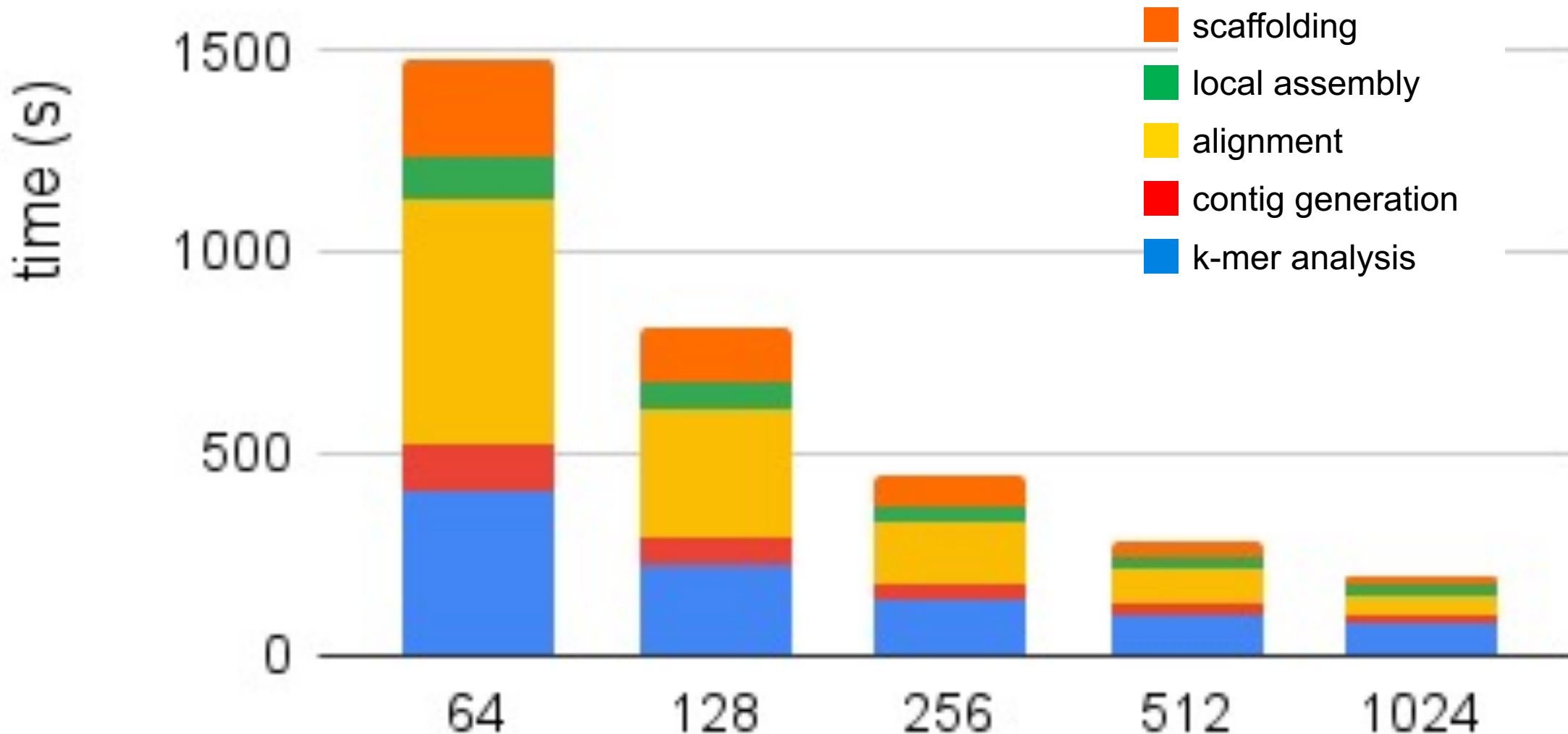
Given sets of strings S and T , find good alignments

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

MetaHipMer Scaling



Running on the Summit system at OLCF with 6 GPUs per nodes

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

MetaHipMer utilized UPC++ features

- C++ templates - efficient code reuse
- dist_object - as a templated functor & data store
- Asynchronous all-to-all exchange - not bulk synchronous
 - 5x improvement at scale relative to previous MPI implementation
- Future-chained workflow
 - Multi-level RPC messages
 - Send by node, then by process
- Promise & fulfill - 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 and support forum

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

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

Exercise Solutions

Solution 1: Ordered hello world

```
int main() {
    upcxx::init();
    for (int i = 0; i < upcxx::rank_n(); ++i) {
        upcxx::barrier();
        if (upcxx::rank_me() == i) {
            std::ofstream fout("output.txt", std::iosbase::app);
            fout << "Hello from process " << upcxx::rank_me()
                << " out of " << upcxx::rank_n() << std::endl;
            sync();
        }
    }
    upcxx::finalize();
}
```

[Link to exercise](#)

Solution 2: Distributed object in Jacobi

Modify the Jacobi code to perform bootstrapping using UPC++ distributed objects (ex2.cpp)

```
global_ptr<double> old_grid_gptr, new_grid_gptr;  
global_ptr<double> right_old_grid, right_new_grid;  
int right; // rank of my right neighbor
```

```
// Obtains grid pointers from the right neighbor and  
// sets right_old_grid and right_new_grid accordingly.
```

```
void bootstrap_right() {  
    dist_object<global_ptr<double>>  
        dobj_old(old_grid_gptr), dobj_new(new_grid_gptr);  
    right_old_grid = dobj_old.fetch(right).wait();  
    right_new_grid = dobj_new.fetch(right).wait();
```

```
    barrier();  
}
```

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

[Link to exercise](#)

Better solution 2: Distributed object in Jacobi

Modify the Jacobi code to perform bootstrapping using UPC++ distributed objects (ex2.cpp)

```
void bootstrap_right() {
    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();
}
```

[Link to exercise](#)

Solution 3: Distributed hash table

Implement the erase and update methods (ex3.hpp)

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

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

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

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

[Link to exercise](#)