BERKELEY LAB

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

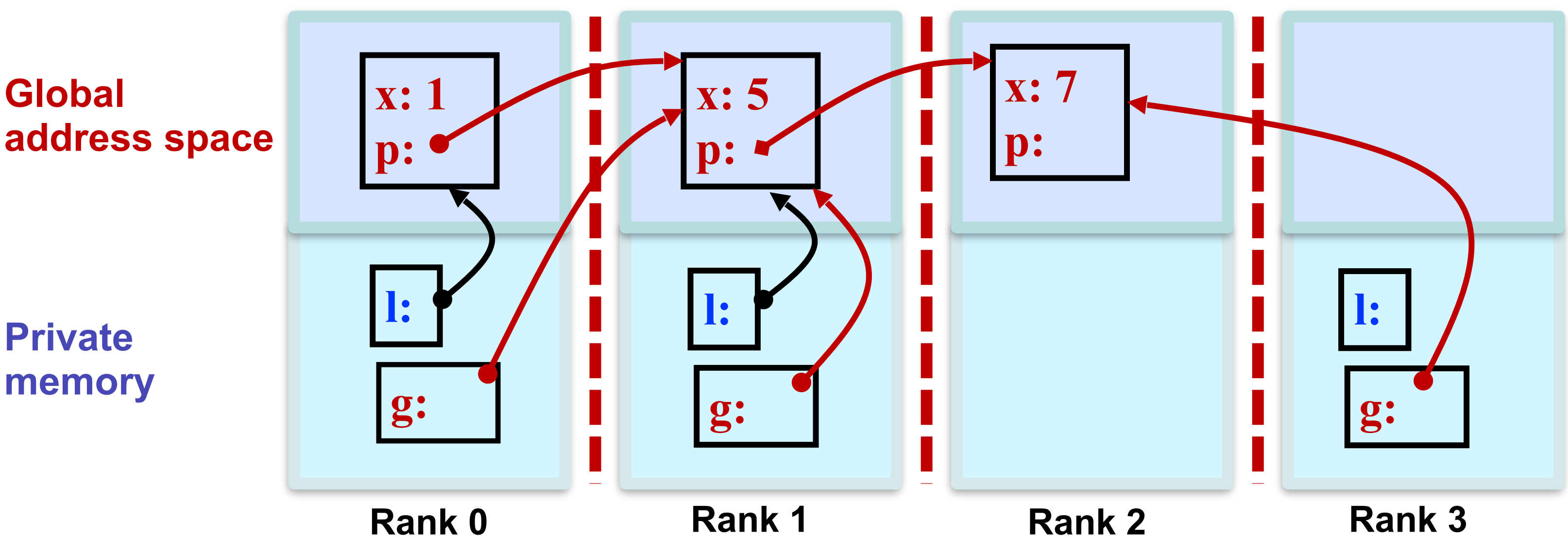
UPC++: A PGAS Library for Exascale Computing

http://upcxx.lbl.gov

Scott B. Baden
Group Lead, Computer Languages and Systems Software Group

UPC++: a C++ PGAS Library

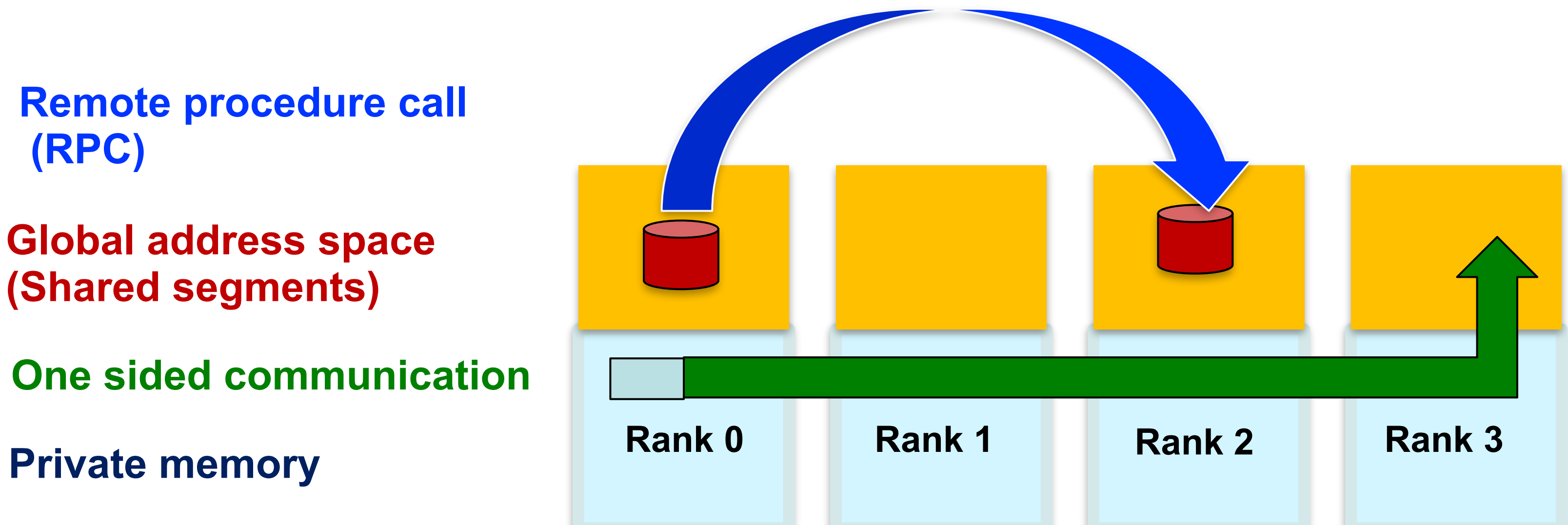
- Global Address Space (P**GAS**)
  - A portion of the physically distributed address space is visible to all processes
- Partitioned (**P**GAS)
  - *Global pointers* to shared memory segments have an *affinity* to a particular rank
  - Explicitly managed by the programmer to optimize for locality



What does UPC++ offer?

- Asynchronous behavior based on futures/promises
  - RMA: Low overhead, zero-copy 1 sided communication. Get/put to a remote location in another address space
  - RPC: Remote Procedure Call: invoke a function remotely A higher level of abstraction, though at a cost

- Design principles encourage performant program design
  - All communication is explicit (unlike UPC)
  - All communication is asynchronous: futures and promises

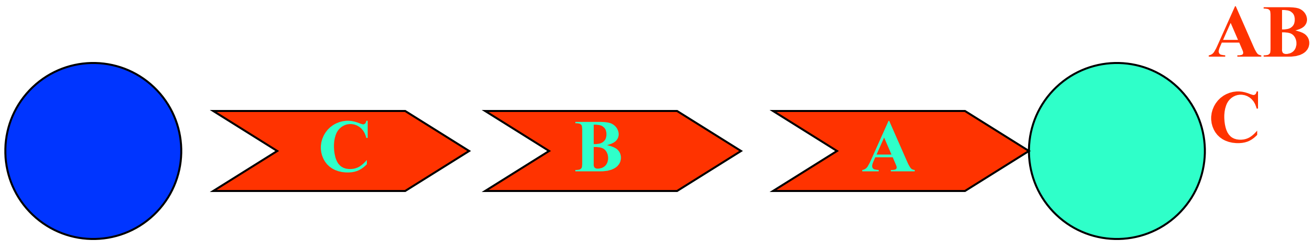


Why is PGAS attractive?

- The overheads are low
  Multithreading can't speed up overhead
- Memory per core is dropping, requiring reduced communication granularity
- Irregular applications exacerbate granularity problem
- Software managed memories are becoming more common, with different access methods.
  We need a unified method for accessing them
- Current and future HPC networks use one-sided transfers at their lowest level and the PGAS model matches this hardware with very little overhead

Where does message passing overhead come from?

- Matching sends to receives
  - Messages have an associated context that needs to be matched to handle incoming messages correctly
  - Data movement and synchronization are coupled
- Ordering guarantees are not semantically matched to the hardware
- UPC++ avoids these factors that increase the overhead
  - No matching overhead between source and target
  - Executes fewer instructions to perform a transfer



BERKELEY LAB

How does UPC++ deliver the PGAS model?

- A “Compiler-Free” approach
  - Need only a standard C++ compiler, leverage C++ standards
  - UPC++ is a C++ template library

- Relies on GASNet-EX for low overhead communication
  - Efficiently utilizes the network, whatever that network may be, including any special purpose offload support

- Designed to allow interoperation with existing programming systems
  - 1-to-1 mapping between MPI and UPC++ ranks
  - OpenMP and CUDA can be mixed with UPC++ in the same way as MPI+X

A simple example of asynchronous execution

By default, all communication ops are split-phased

- Initiate operation
- Wait for completion

  A future holds a value and a state: ready/not ready

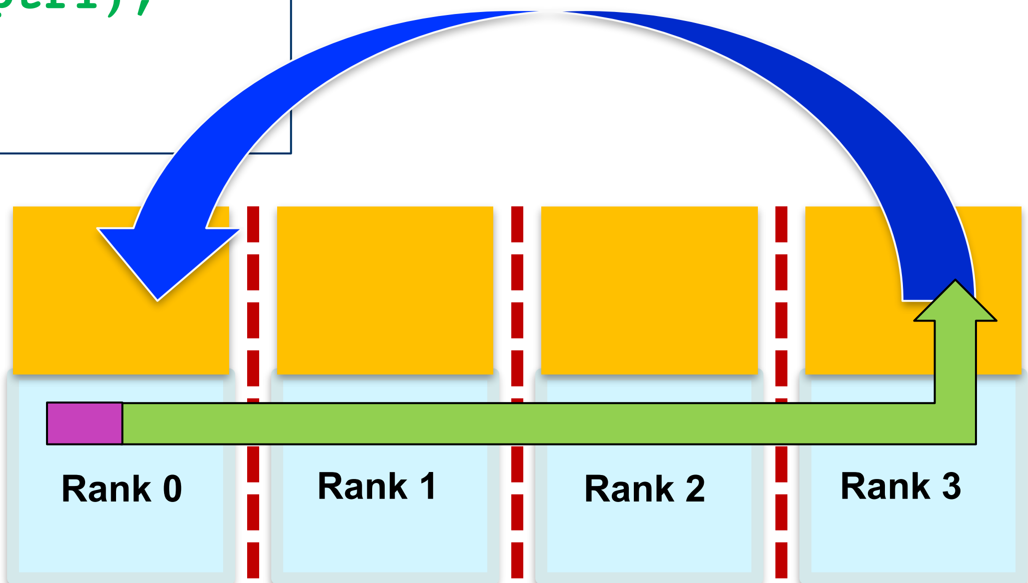
```
global_ptr<T> gptr1 = . . .;
future<T> f1 = rget(gptr1);
// unrelated work...
T t1 = f1.wait();
```

Wait returns with result when rget completes

Global address space

Start the get

Private memory

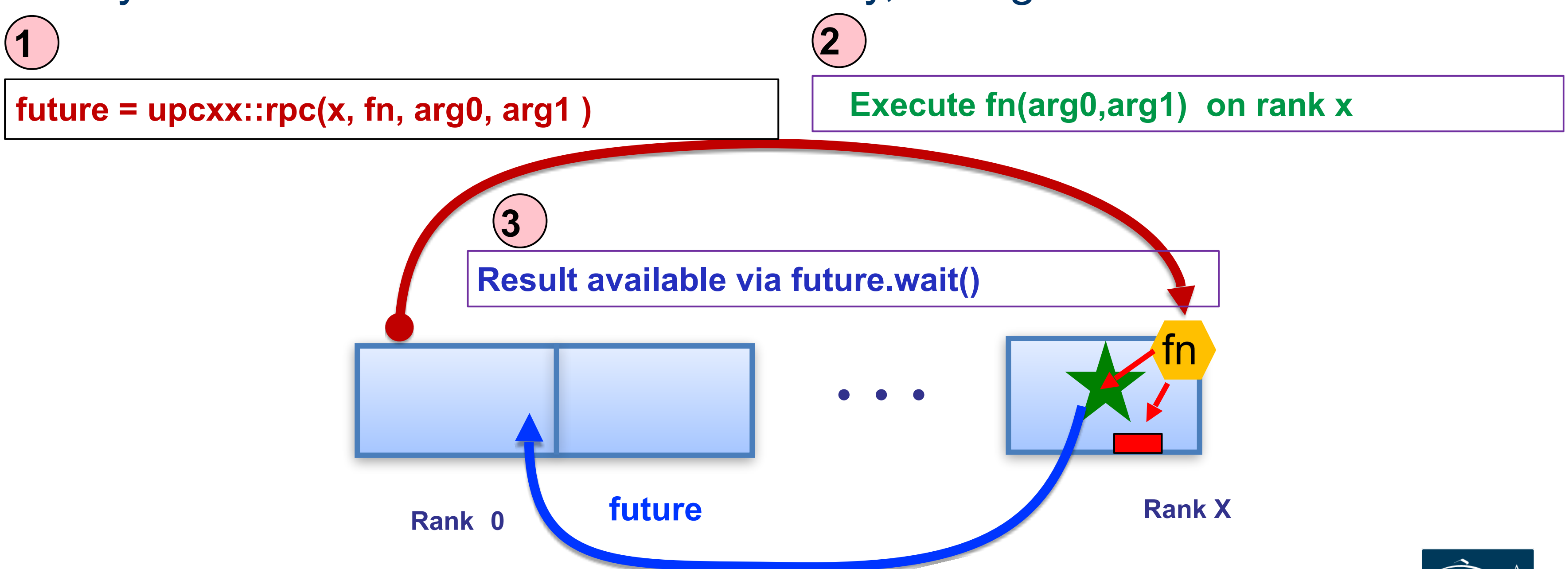


Simple example of remote procedure call

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

1. Injects the RPC to the *target rank* X
2. Executes fn(key, arg0, arg1) on target rank at some future time determined at the target
3. Result becomes available to the caller via the future

Many invocations can run simultaneously, hiding data movement



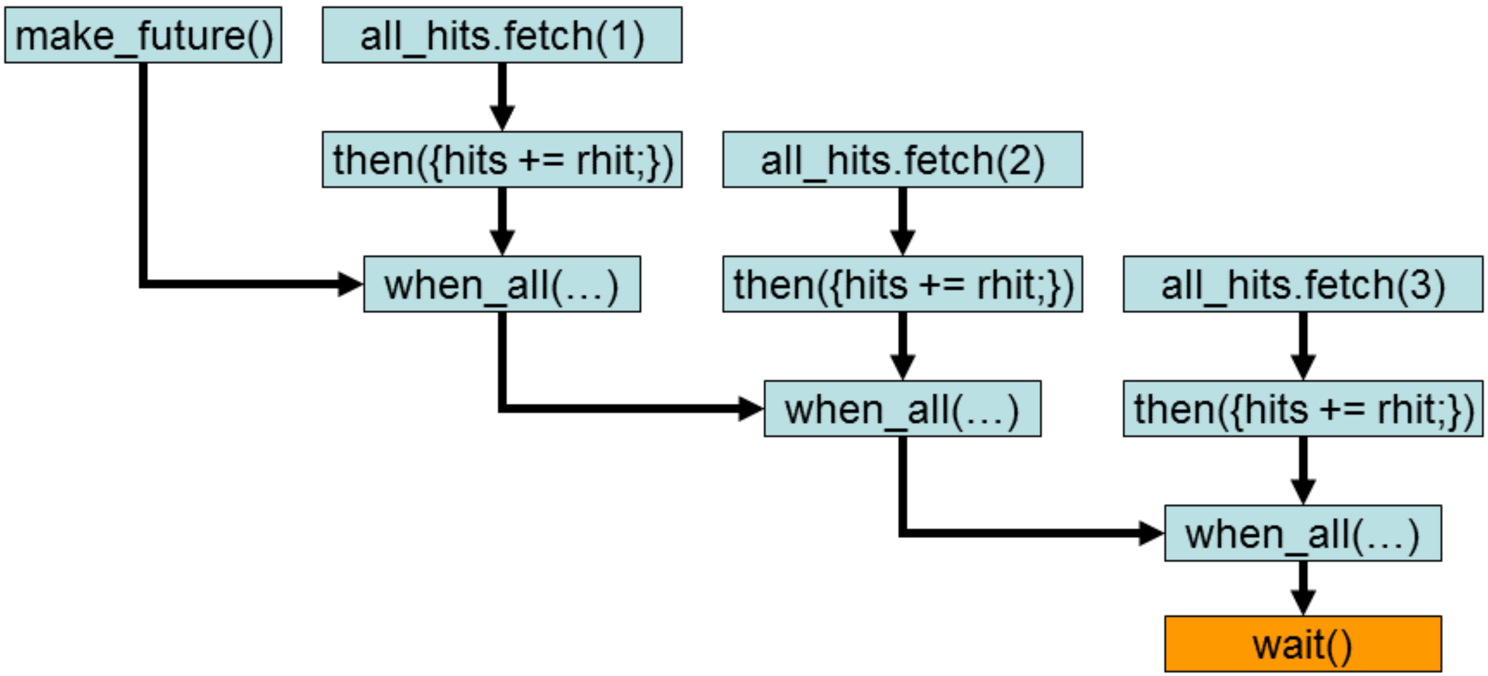
Composing asynchronous operations

- Rput, rget and RPCs return a future
- We can build a DAG of futures, synchronize on the whole rather than on the individual operations
  - Attach a callback: **`.then(Foo)`**
  - **`Foo`** is the completion handler, a function or λ
    - runs locally when the **`rget`** completes
    - receives arguments containing result associated with the future

```
double Foo(int x){ return sqrt(2*x); }
global_ptr<int> gptr1 = ...;
future<int>    f1 = rget(gptr1);
future<double> f2 = f1.then(Foo);
// DO SOMETHING ELSE
double y = f2.wait();
```

Conjoining futures

- We can join futures using when_all()
- (Example taken from *UPC++ Programmer's Guide*)



Road Map

Application proxies

Library Performance

Other features of UPC++

Application: *De Novo* Genome Assembly

Construct a genome (chromosome) from a pool of short fragments, called *reads*, produced by sequencers

Analogy: shred many copies of a book, and reconstruct the book by examining the pieces

Complications: shreds of other books may be intermixed, can also contain errors

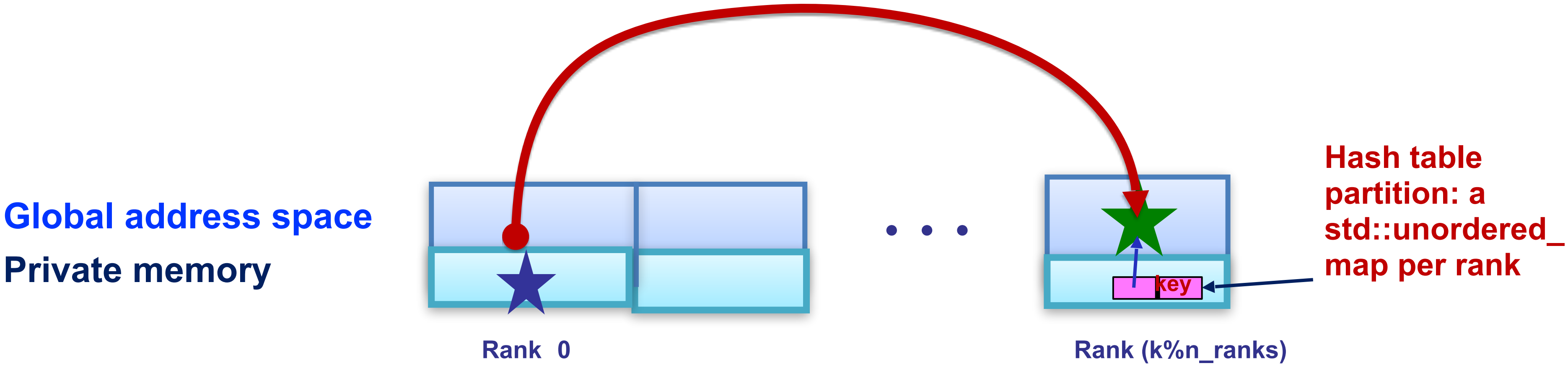
Chop the reads into fixed-length fragments (k-mers)

K-mers form a De Bruijn graph, traverse the graph to construct longer sequences

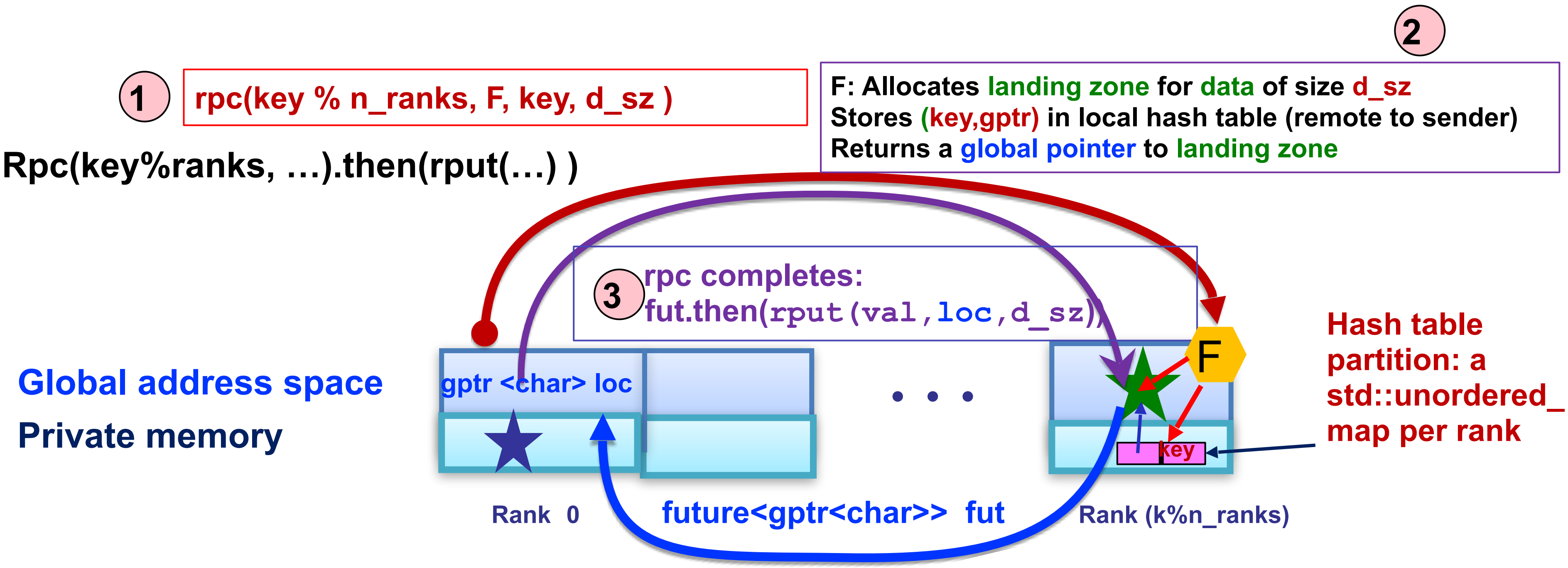
Graph is stored in a *distributed hash table*

Distributed hash table implementation

- **Used in de-novo Genome Assembly**
- **This example motivates Remote Procedure Call (RPC)**
- **RPC** simplifies the distributed hash table design
- Store value in a distributed hash table, at a remote location



Distributed hash table implementation



- RPC inserts the key @ target and obtains a landing zone pointer
- Once the RPC completes, an attached callback (.then) uses rput to store the associated data
- We use futures to build a small chain of dependent operations
- The returned future represents the whole operation

The hash table code

- We use lambda for the RPC function in this example
- RPC inserts key meta data at the target, & allocates the landing zone
  - Leverage implicit synchronization of rpc execution
- Once the RPC completes, the callback (.then()) that was attached to the RPC uses a zero copy rput to store the associated data
  - Exploits the power of rput for high performance RMA where available

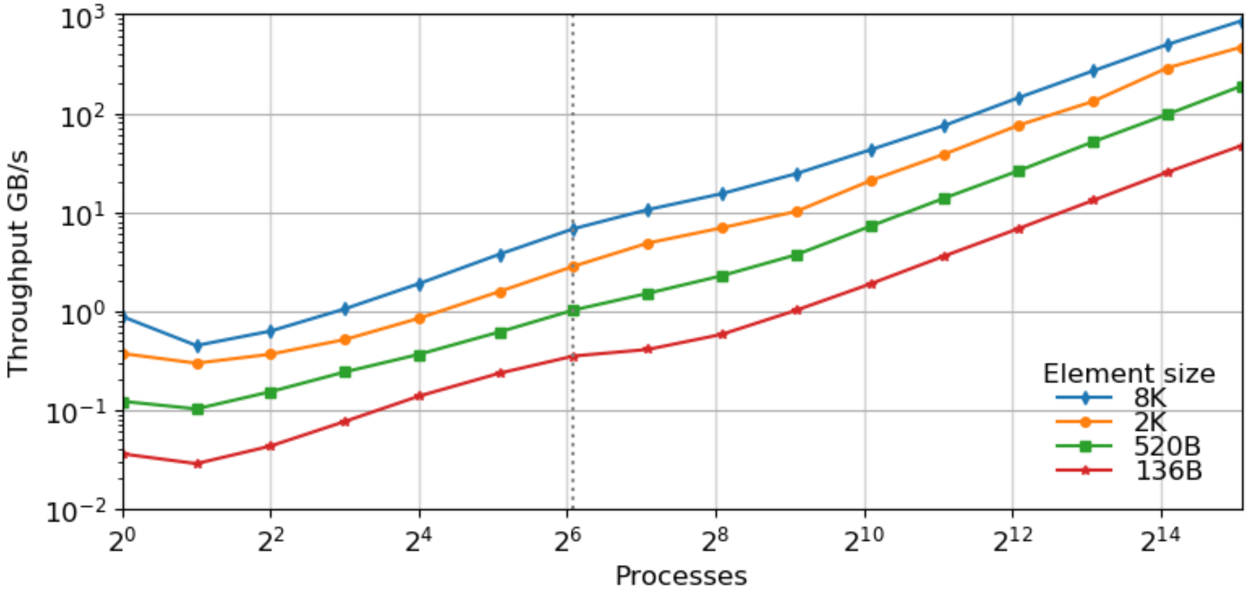
```
// C++ global variables correspond to rank-local state
std::unordered_map<uint64_t, global_ptr<char> > local_map;
// insert a key-value pair and return a future
future<> dht_insert(uint64_t key, char *val, size_t d_sz) {
   auto f1 = rpc(key % rank_n(),  // RPC obtains location for the data
          [](uint64_t key, size_t d_sz) -> global_ptr<char> {
             global_ptr<char> gptr = new_array<char>(d_sz);
             local_map[key] = gptr;             // insert in local map
             return gptr;
          }, key, d_sz);
   return f1.then( // callback executes when RPC completes
          [val,d_sz](global_ptr<char> loc) -> future<> { // λ: RMA put
                return rput(val, loc, d_sz); }
   );
}
```

λ function

λ for callback

Benefits of UPC++: distributed hash table

- Randomly distributed keys
- Excellent weak scaling up to 32K cores
- RPC leads to simplified and more efficient design
  - Key insertion and storage allocation handled at target
  - Without RPC, complex updates would require explicit synchronization and the need to use global storage, e.g. OpenSHMEM and MPI one-sided



Cori @ NERSC (KNL)

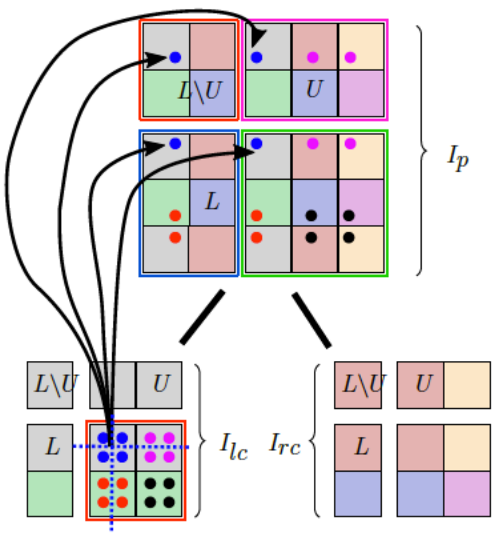
Cray XC40

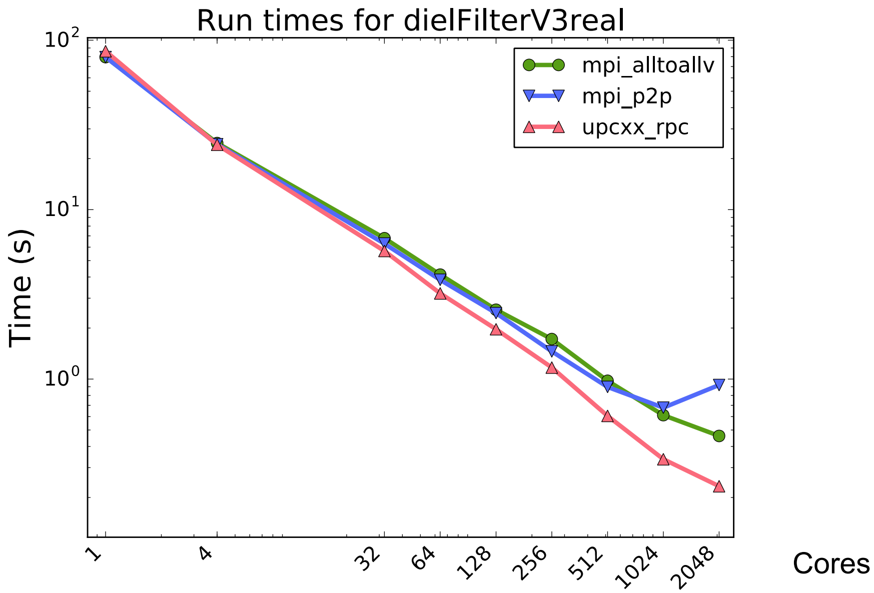
The productivity benefit of RPC

- More natural way to express hash table insertion with RPC than with one sided communication or message passing
- RPC encapsulates argument passing, queue management and progress, factoring them out of the application code
- More generally, RPC simplifies the coding in updating complicated distributed data structures

UPC++ improves sparse solver performance

- Sparse matrix factorizations have low computational intensity and irregular communication patterns
- Extend-add operation is an important building block for multifrontal sparse solvers
- RPC sends child contributions to the parent
- RPC vital to improving performance





Strong scaling on Cori-KNL @ NERSC (extended add time only)

Road Map

Applications Proxies

Library performance

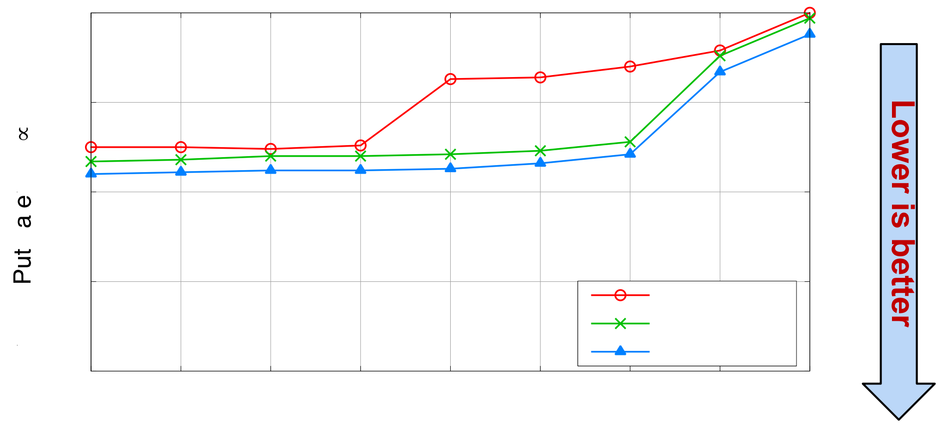
Other features of UPC++

A look under the hood of UPC++

- Relies on GASNet-EX to provide low overhead communication
- Efficiently utilizes the network, whatever that network may be, including any special purpose support - low overheads
- Get/put map directly onto the network hardware's global address support, when available
- Data movement has a low overhead because there is no matching of sender to receiver
- RPC uses an active message (AM) to enqueue the function handle remotely. Any return result is also transmitted via an AM
- RPCs can only make progress inside a call to a UPC++ method (Also a distinguished progress() method)
- Thus, RPCs are serialized at the target, and this attribute can be used to avoid explicit synchronization

Performance of UPC++ - Latency

- Ping pong microbenchmark using blocking RMA
  - ~20% latency improvement from 16 to 256 bytes
  - Lines never cross at long message sizes not shown
- UPC++ rput, GASNet-Ex *testsmall* and publicly available MPI/ IMB-RMA benchmark suite
  - Long sequence of blocking operations:
    issue put, wait for remote completion, repeat…
  - Latency = average time



Cori I @ NERSC (Haswell)

Cray XC40

Road Map

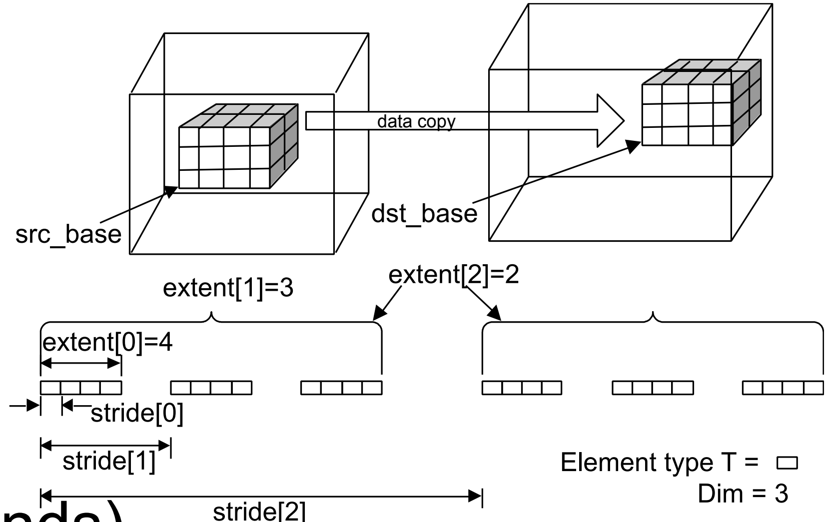
Application Proxies

Library performance

Other features of UPC++

Other features

- Completions
  - Know when the source memory can be modified, when the op has completed at the target
- Remote Atomics
- Non-contiguous transfers
- Distributed Objects
- Collectives (ongoing)
- Teams
- GPU memory (memory kinds)
  - Uniform interface to host and device memory
  - NEW to the latest release March 15, 2019



Toward distributed data structures

- Other models (UPC, CAF, OpenSHMEM) implement shared arrays via a symmetric heap
  - Scalable and portable implementation is difficult
  - Requires globally collective allocation that does not compose with subset teams
- UPC++ doesn't have a symmetric heap:
  heap allocation is not collective
- Initially, each rank only knows the locations of shared objects that it allocated
- How does a rank learn the locations of shared objects allocated by other ranks?

Distributed objects in UPC++

- How does a rank learn the locations of shared objects allocated by other ranks?
- UPC++ provides the *distributed object* (like co-arrays)
  - Globally unique name for each distributed object
  - Each entry holds a rank-specific value
  - Retrieve a remote value using a rank ID
  - Can be used to build a scalable, globally visible directory
- Distributed objects can be used to build shared distributed arrays, among other data structures
- UPC++ does not prescribe solutions, rather it provides building blocks for constructing them

Distributed 1D Arrays over UPC++

- Design is a work in progress
  - Likely similar to UPC pure-blocked shared array
  - Will support dynamic length and block size
  - Will be built over distributed objects, so it will have a scalable representation and support subset teams

```
dist_array<double> array(N, some_team);
// fetch ith element of array
future<global_ptr<double>> fut1 = array.pointer_to(i);
future<double> fut2 = fut1.then(rget);
// ... other work
double val = fut2.wait();
```

UPC++ in context

- Only existing library with PGAS support that also offers RPC (X10, Chapel and Habanero are languages)
- OpenSHMEM is considering adding RPC
- Besides DASH, only model that support subset teams
- UPC++ supports distributed objects, a generalization of distributed arrays
  - Can construct an object over a subset team
  - Avoids a symmetric heap, which is not scalable
  - Distributed arrays: UPC, OpenSHMEM, DASH, X10, Chapel, co-array C++ via co-arrays

UPC++ = Productivity + Performance

Productivity

- UPC++ does not prescribe solutions for implementing distributed irregular data structures: it provides building blocks
- Interoperates with MPI, OpenMP and CUDA
- Develop incrementally, enhance selected parts of the code

Reduced communication costs

- Embraces communication networks that use one-sided transfers at their lowest level
- Low overhead reduces the cost of fine-grained communication
- Overlap communication via asynchrony and futures

Summary

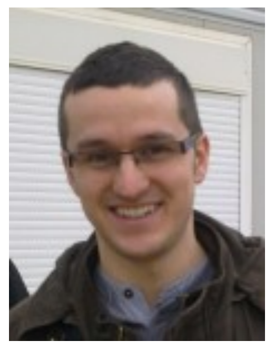
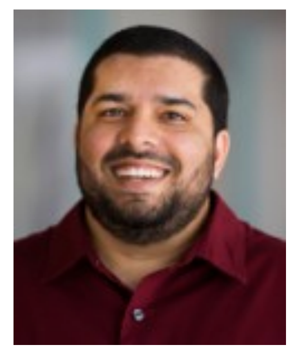
- UPC++ provides future and continuation-based completion handling; remote procedure calls; one sided communication
  - Delivers close-to-the-metal performance in RMA communication using GASNet-EX
  - Overlap communication via asynchronous execution
  - Use network RDMA capability to support a programming model that combines explicit locality control with shared memory
- More advanced constructs (not discussed)
  - Remote atomics
  - Distributed objects, teams and collectives
  - Promises, personas (end points), generalized completion
  - Serialization
  - Memory kinds for GPU memory (coming)

The Pagoda Team

- Scott B. Baden (PI)
- Paul H. Hargrove (co-PI)
- John Bachan
- Dan Bonachea
- Steve Hofmeyr
- Mathias Jacquelin
- Amir Kamil
- Hadia Ahmed
- Alumni:
  Brian van Straalen, Khaled Ibrahim






Code and documentation at http://upcxx.lbl.gov

Acknowledgements

Early work with UPC++ involved Yili Zheng, Amir Kamil, Kathy Yelick, and others [IPDPS '14]

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), funded by the U.S. Department of Energy

ECP partners: ExaBiome- Kathy Yelick, Sparse Solvers – Sherry Li and Pieter Ghysels, AMREx – John Bell and Tan Nguyen [LBNL]

Academia: Alex Pöppl and Michael Bader (TUM) – Actor framework
Niclas Jansson and Johann Hoffman (KTH) – unstructured flow solvers

https://tinyurl.com/y79tab2n