BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

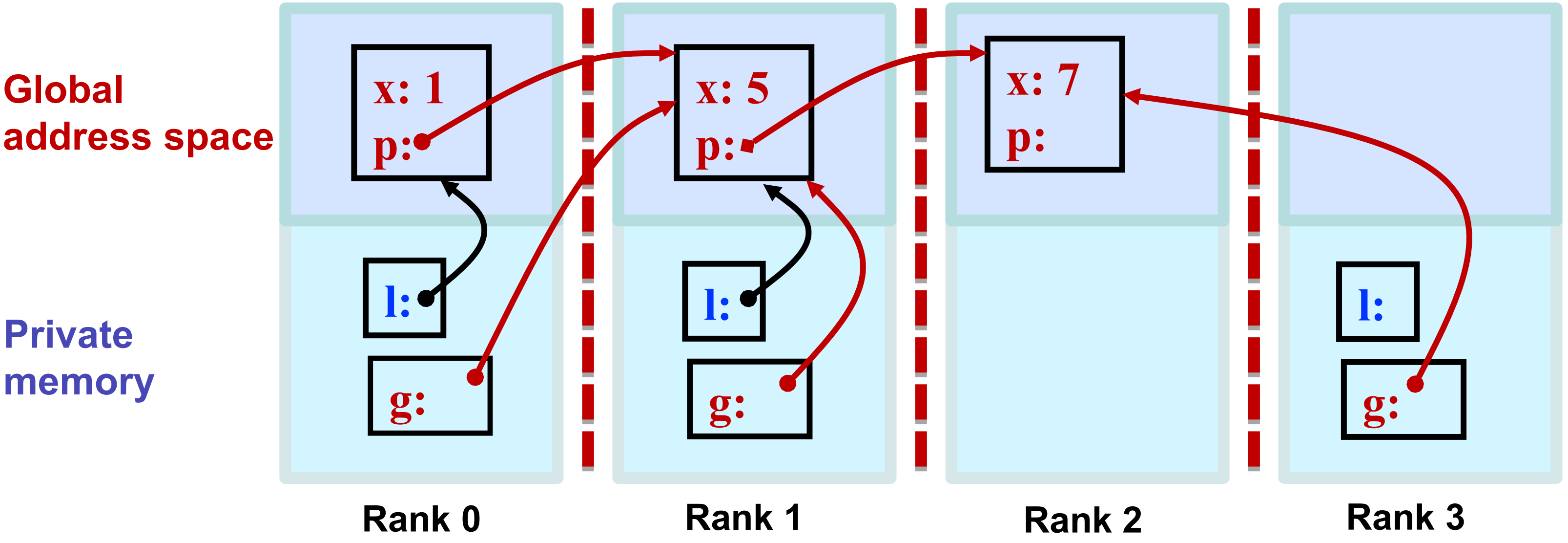
UPC++: A High-Performance Communication Framework for Asynchronous Computation

John Bachan, Scott B. Baden, Steven Hofmeyr, **Mathias Jacquelin**
Amir Kamil, Dan Bonachea, Paul H. Hargrove, Hadia Ahmed

Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California, USA

UPC++: a C++ PGAS Library

- Global Address Space (P**GAS**)
  - A portion of the physically distributed address space is visible to all processes. Now generalized to handle GPU memory
- 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



Why is PGAS attractive?

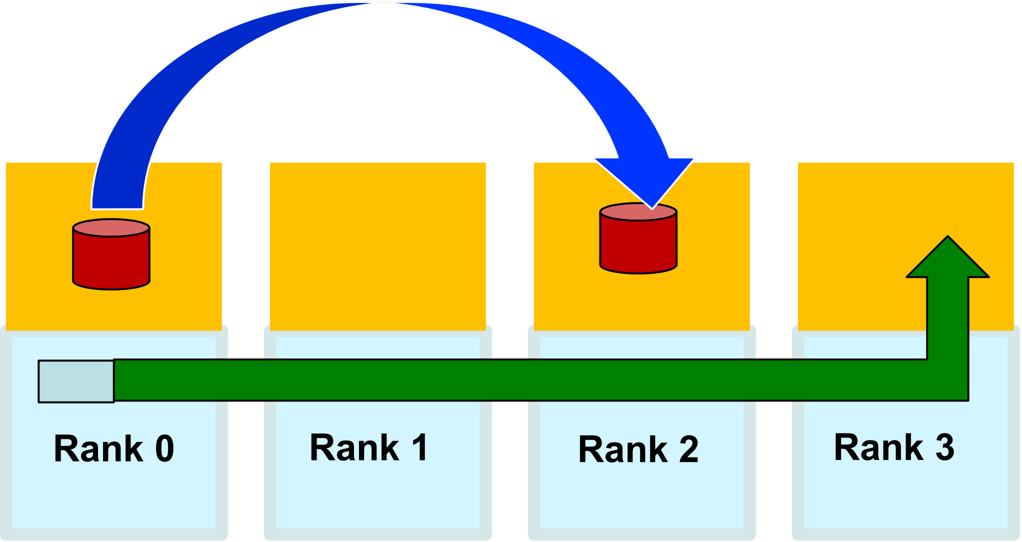
- **The overheads are low**
  Multithreading can’t speed up overheads
- Memory-per-core is dropping, requiring **reduced communication granularity**
- Irregular applications exacerbate granularity problem

Asynchronous computations are critical

- Current and future HPC networks use one-sided transfers at their lowest level and the PGAS model matches this hardware with very little overhead

What does UPC++ offer?

- Asynchronous behavior based on futures/promises
  - **RMA**: Low overhead, zero-copy one-sided communication. Get/put to a remote location in another address space
  - **RPC: Remote Procedure Call**: invoke a function remotely A higher level of abstraction, though at a cost
- Design principles encourage performant program design
  - All communication is syntactically explicit (unlike UPC)
  - All communication is asynchronous: futures and promises
  - Scalability



Remote procedure call (RPC)

Global address space (Shared segments)

One sided communication

Private memory

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

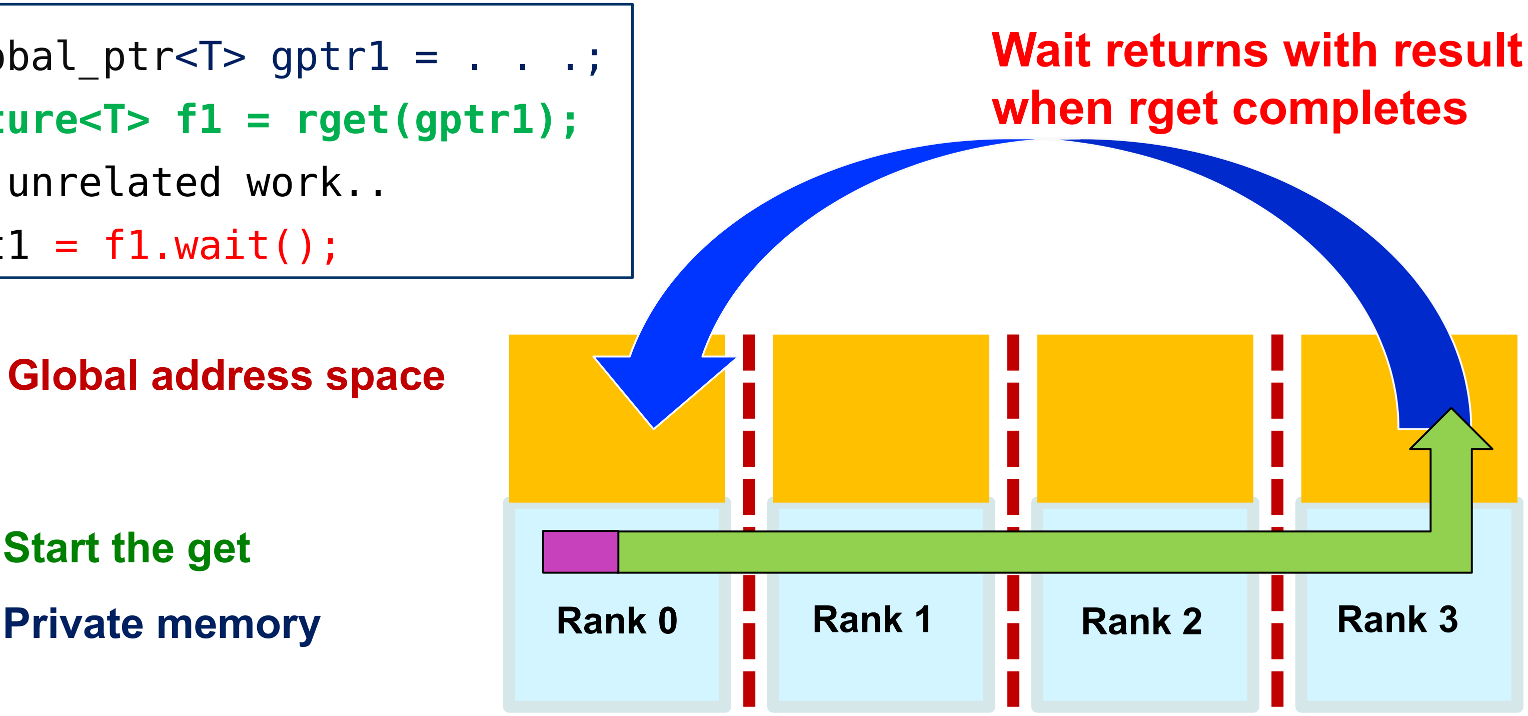
A simple example of asynchronous execution

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();
```

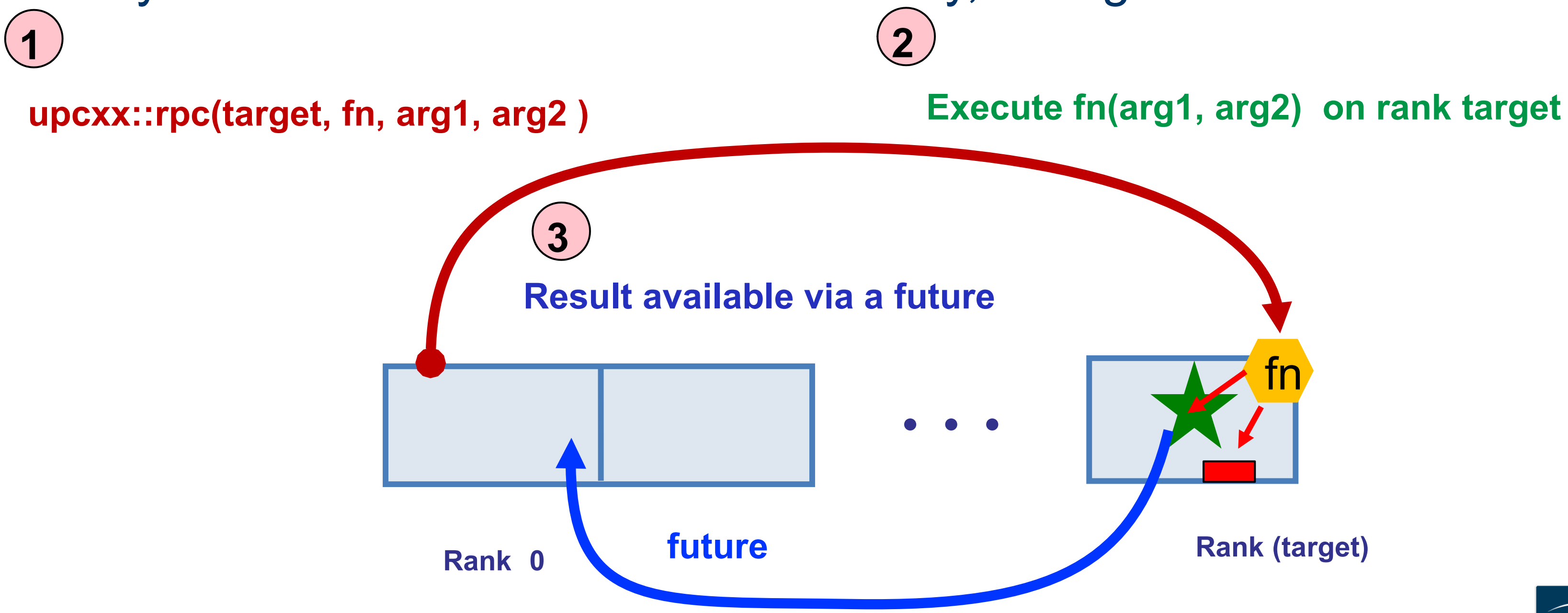


Simple example of remote procedure call

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

1. Injects the RPC to the *target* rank
2. Executes fn(arg1, arg2) 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



①

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

②

Execute fn(arg1, arg2) on rank target

③

Result available via a future

Asynchronous operations

- **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;
// …  gptr1 initialized
future<int>    f1 = rget(gptr1);
future<double> f2 = f1.then(Foo);
// DO SOMETHING ELSE
double y = f2.wait();
```

A look under the hood of UPC++

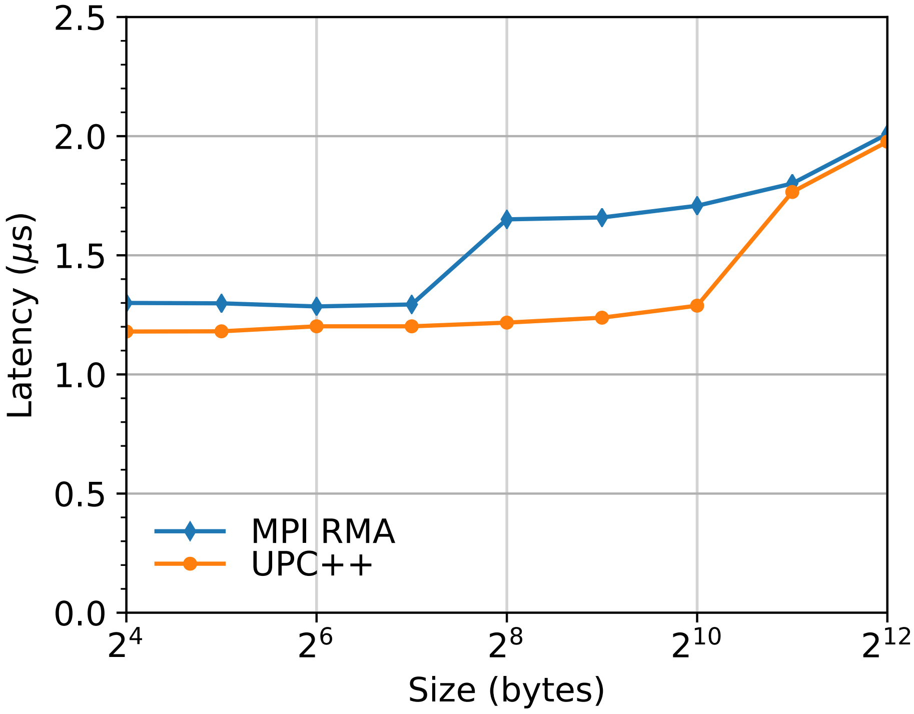
- Relies on GASNet-EX to provide low-overhead communication
  - Efficiently utilizes the network, whatever that network may be, including any special-purpose support
  - Get/put map directly onto the network hardware’s global address support, when available
- RPC uses an active message (AM) to enqueue the function handle remotely.
  - Any return result is also transmitted via an AM
- RPC callbacks are only executed inside a call to a UPC++ method (Also a distinguished progress() method)
  - RPC execution is serialized at the target, and this attribute can be used to avoid explicit synchronization

https://gasnet.lbl.gov

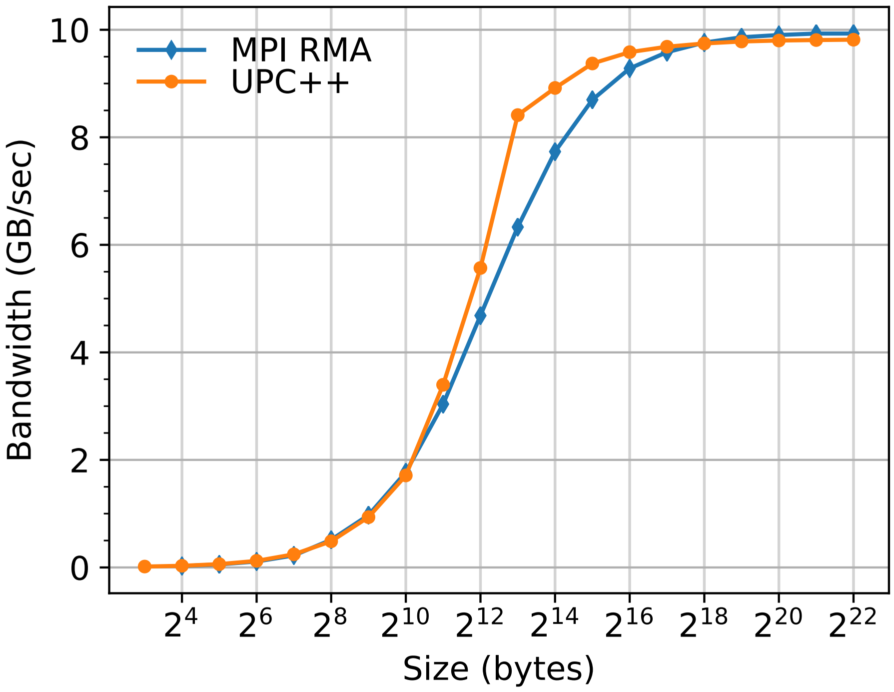
RMA microbenchmarks

Experiments on NERSC Cori:
- Cray XC40 system

- Two processor partitions:
  - Intel Haswell (2 x 16 cores per node)
  - Intel KNL (1x68 cores per node)



Round-trip Put Latency (lower is better)

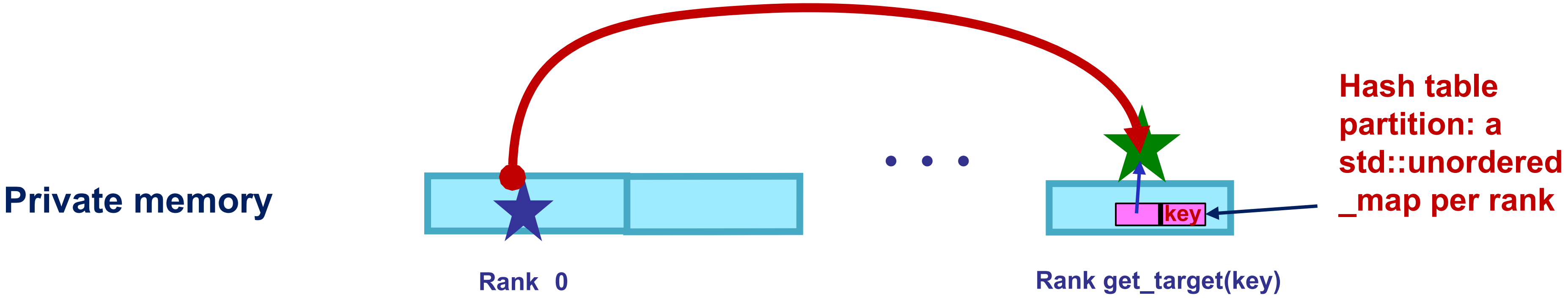


Flood Put bandwidth (higher is better)

Data collected on Cori Haswell

Distributed hash table – Productivity

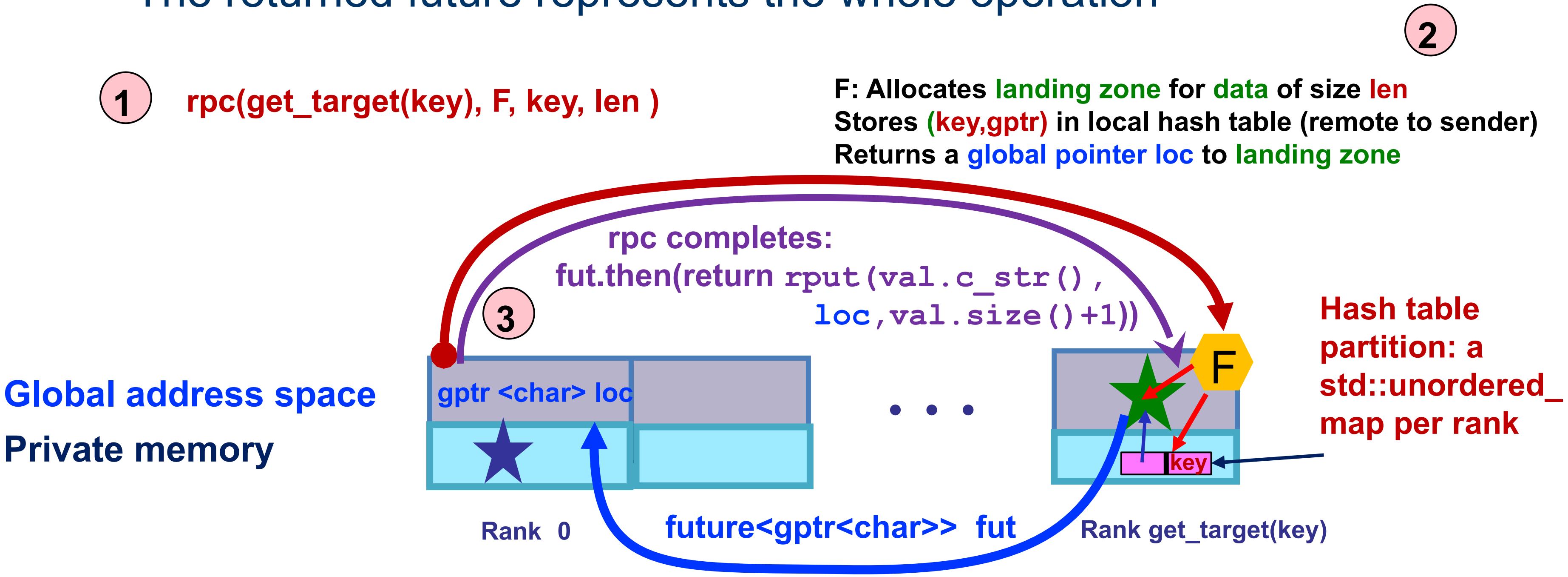
- **Uses Remote Procedure Call (RPC)**
- **RPC** simplifies the distributed hash table design
- Store value in a distributed hash table, at a remote location



```
// C++ global variables correspond to rank-local state
std::unordered_map<string, string > local_map;
// insert a key-value pair and return a future
future<> dht_insert(const string & key, const string & val ) {
   return upcxx::rpc(get_target(key),
       [](string key, string val) {
          local_map.insert ({key ,val });
       }, key, val);
}
```

Distributed hash table – Performance

- RPC+RMA implementation, higher performance (zero-copy)
- RPC inserts the key at target and obtains a landing zone pointer
- Once the RPC completes, an attached callback (.then) uses **zero-copy rput** to store the associated data
- The returned future represents the whole operation



The hash table code

```
// C++ global variables correspond to rank-local state
std::unordered_map<string, global_ptr<char> > local_map;
// insert a key-value pair and return a future
future<> dht_insert(const string & key, const string & val) {
   auto f1 = rpc( get_target(key),  // RPC obtains location for the data
```

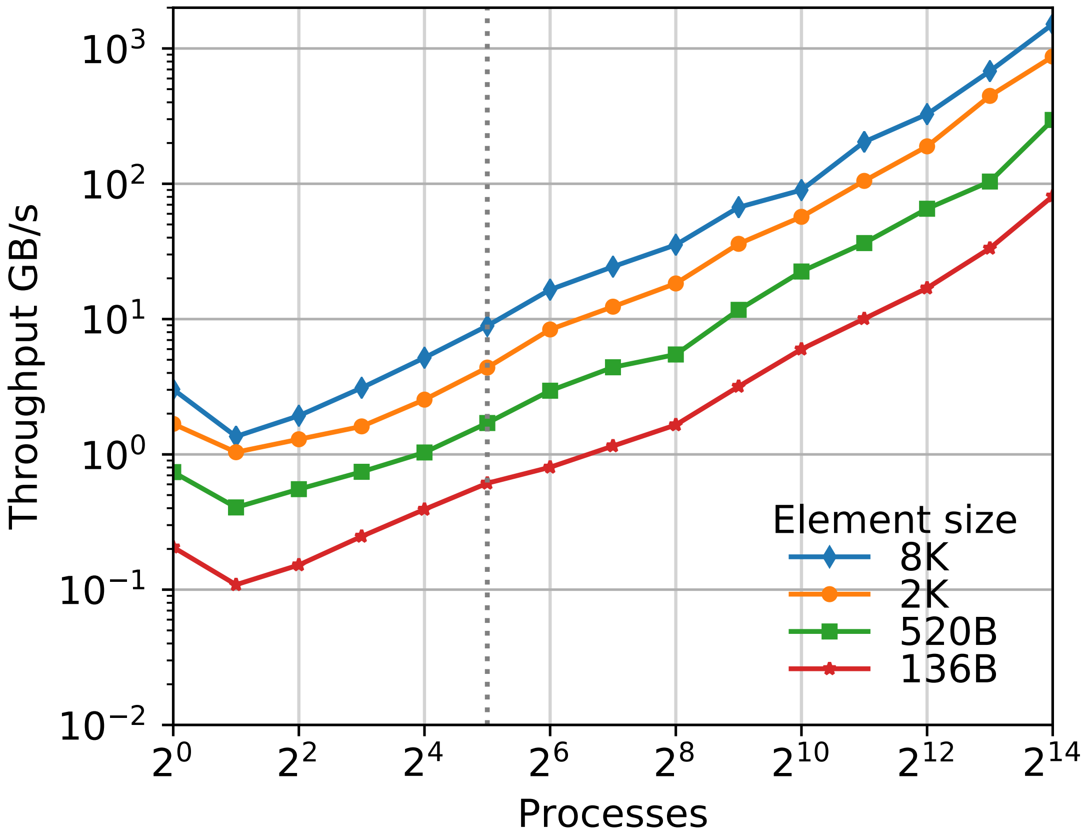
λ function

```
              [](string key, size_t len) -> global_ptr<char> {
                 global_ptr<char> gptr = new_array<char>(len);
                 local_map[key] = gptr;             // insert in local map
                 return gptr;
                   }, key, val.size()+1 );
   return f1.then( // callback executes when RPC completes
```

λ for callback

```
               [val](global_ptr<char> loc) -> future<> { // : RMA put
                        return rput(val.c_str(), loc, val.size()+1); }
   );
}
```

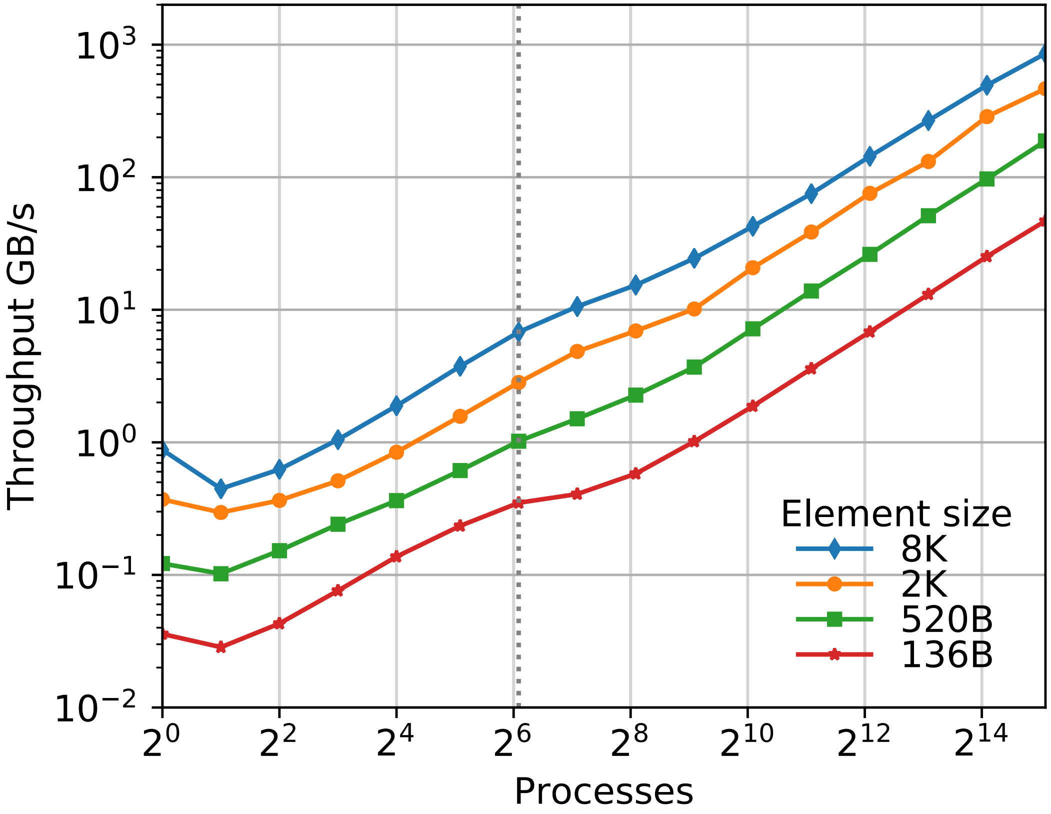
Weak scaling of distributed hash table insertion



- Randomly distributed keys
- Excellent weak scaling up to 32K cores
- RPC leads to simplified and more efficient design
- RPC+RMA achieves high performance at scale

NERSC Cori Haswell

Weak scaling of distributed hash table insertion



- Randomly distributed keys
- Excellent weak scaling up to 32K cores
- RPC leads to simplified and more efficient design
- RPC+RMA achieves high performance at scale

NERSC Cori KNL

UPC++ improves sparse solver performance

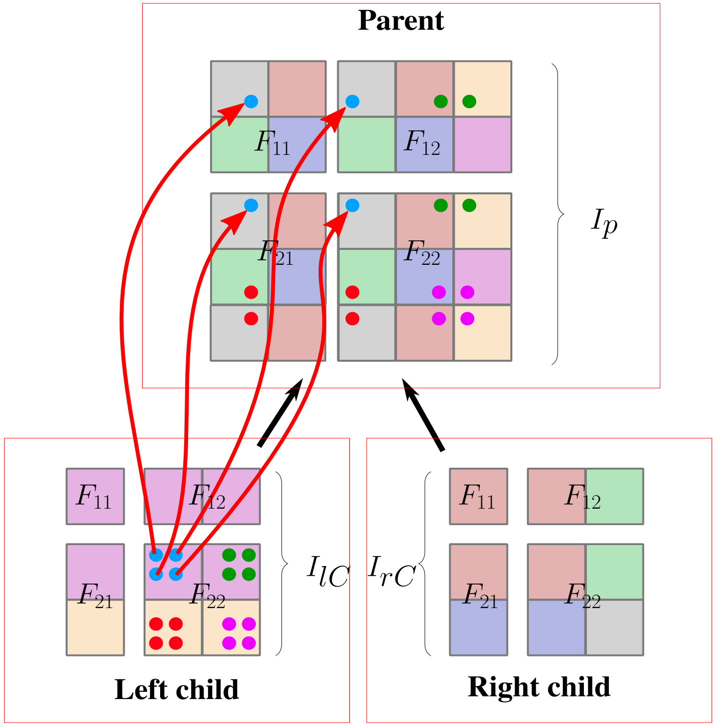
- Sparse matrix factorizations have low computational intensity and irregular communication patterns

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

- Sparse factors are organized as a hierarchy of condensed matrices called **frontal matrices:**
  - 4 sub-matrices:

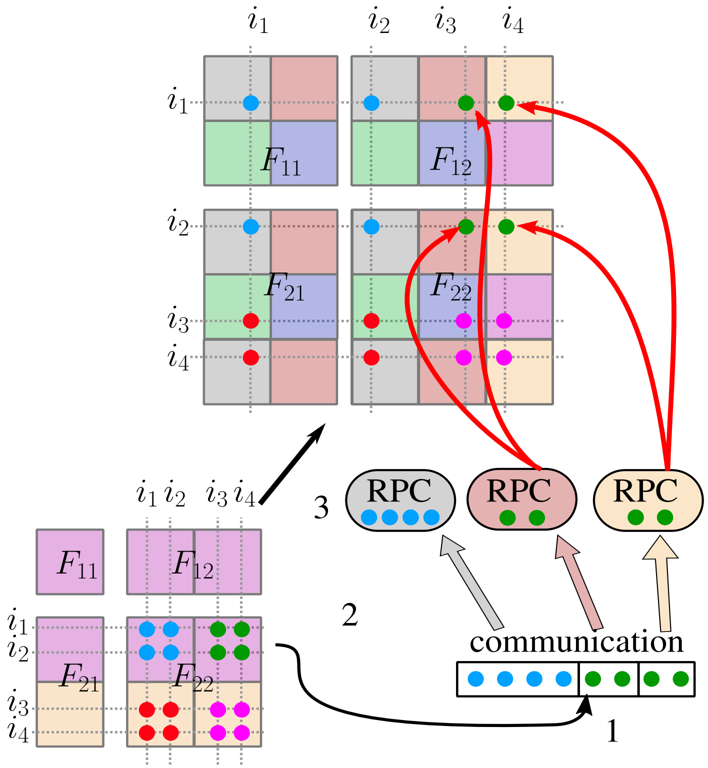
    factors + contribution block

  - Contribution blocks are **accumulated in parent**

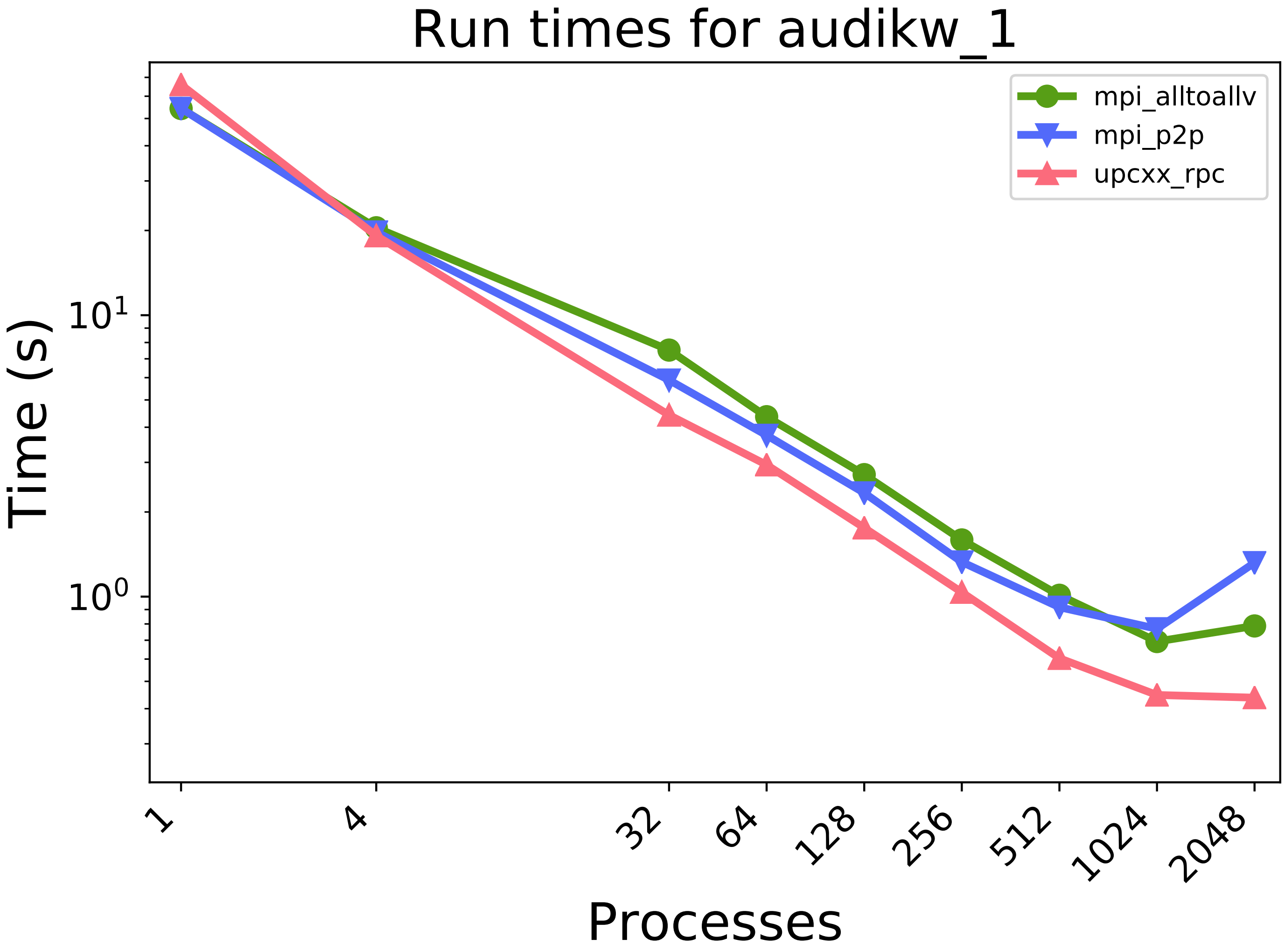


UPC++ improves sparse solver performance

- Data is packed 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
  - **RPC compare indices and accumulate contributions on the target**

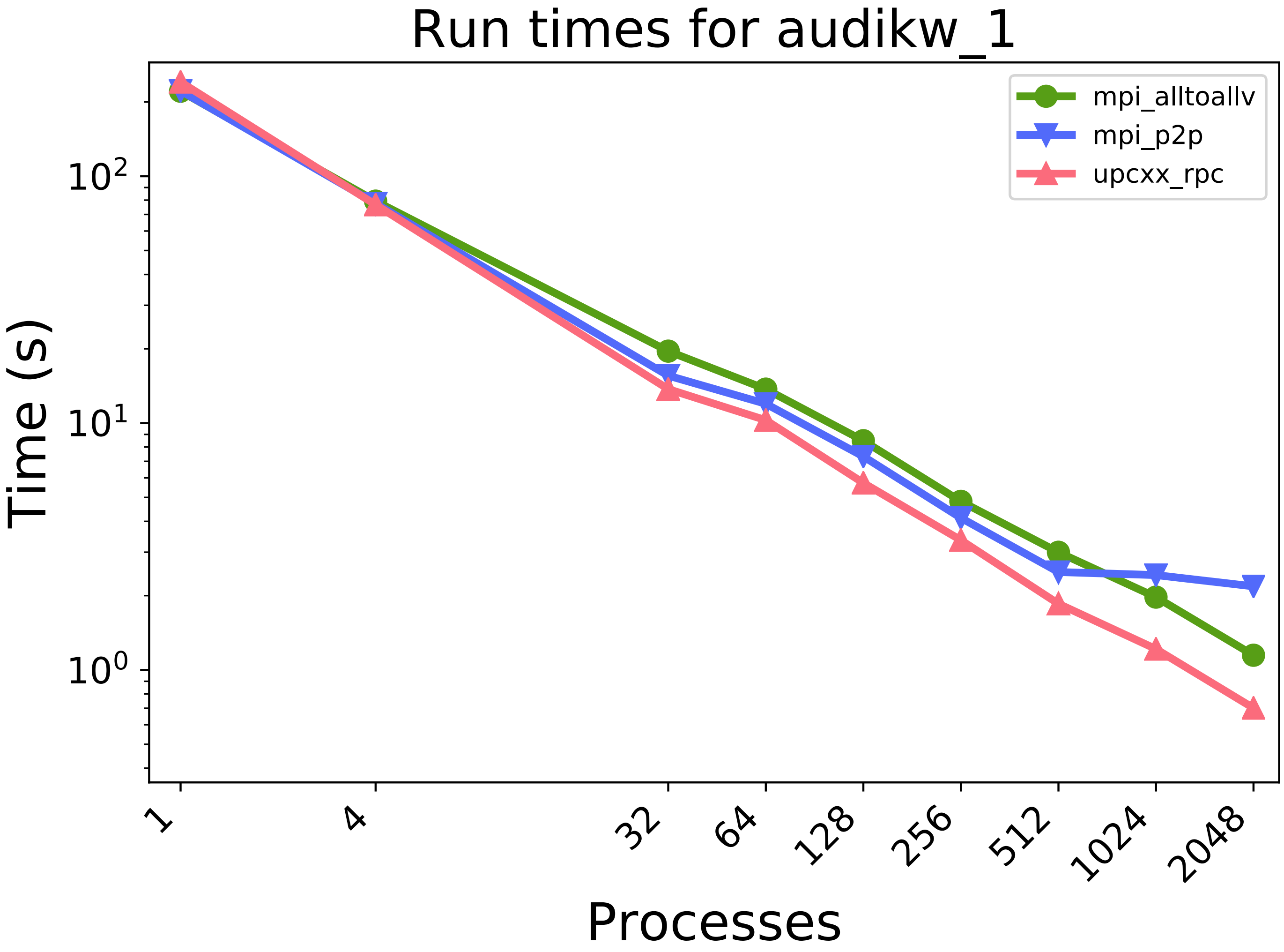


UPC++ improves sparse solver performance



Assembly trees / Frontal matrices extracted from STRUMPACK

UPC++ improves sparse solver performance



Assembly trees / Frontal matrices extracted from STRUMPACK

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

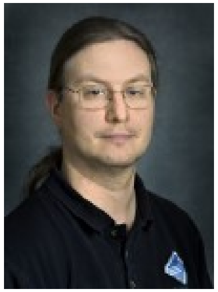
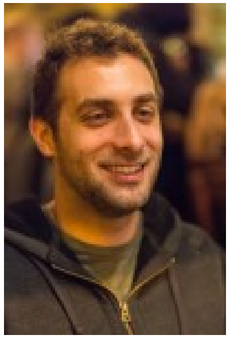
Reduced communication costs

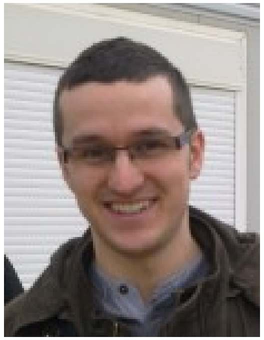
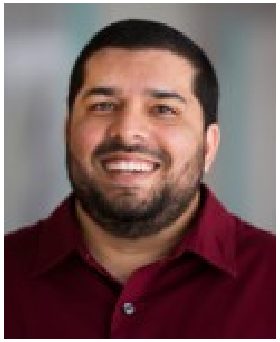
- Embraces communication networks that use one-sided transfers at their lowest level
- Low overhead reduces the cost of fine-grained communication
- Overlap communication via asynchrony and futures
- High-performance distributed hash table
- Increased efficiency in the extend-add operation (sparse solvers)

More advanced constructs (not discussed)

- Remote atomics, distributed objects, teams and collectives
- Promises, end points, generalized completion
- Serialization, non-contiguous transfers

The Pagoda Team


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

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

Examples and extras available at the end of May

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 collaborators: Kathy Yelick, Sherry Li, Pieter Ghysels, John Bell and Tan Nguyen (Lawrence Berkeley National Laboratory)

Academic collaborators: Alex Pöppl and Michael Bader (TUM) ,
Niclas Jansson and Johann Hoffman (KTH),
Sergio Martin (ETH-Zurich),
Phuong Ha (Arctic Univ. of Norway)

http://upcxx.lbl.gov

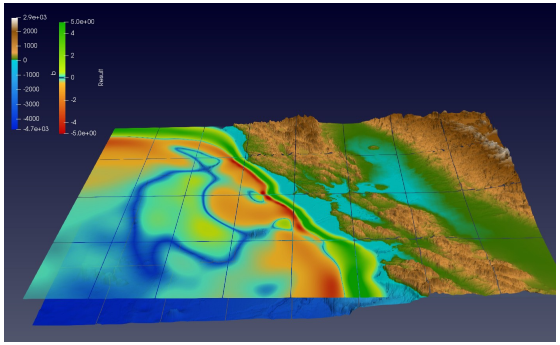


Figure courtesy Alexander Pöppl

Additional information

Related work on PGAS

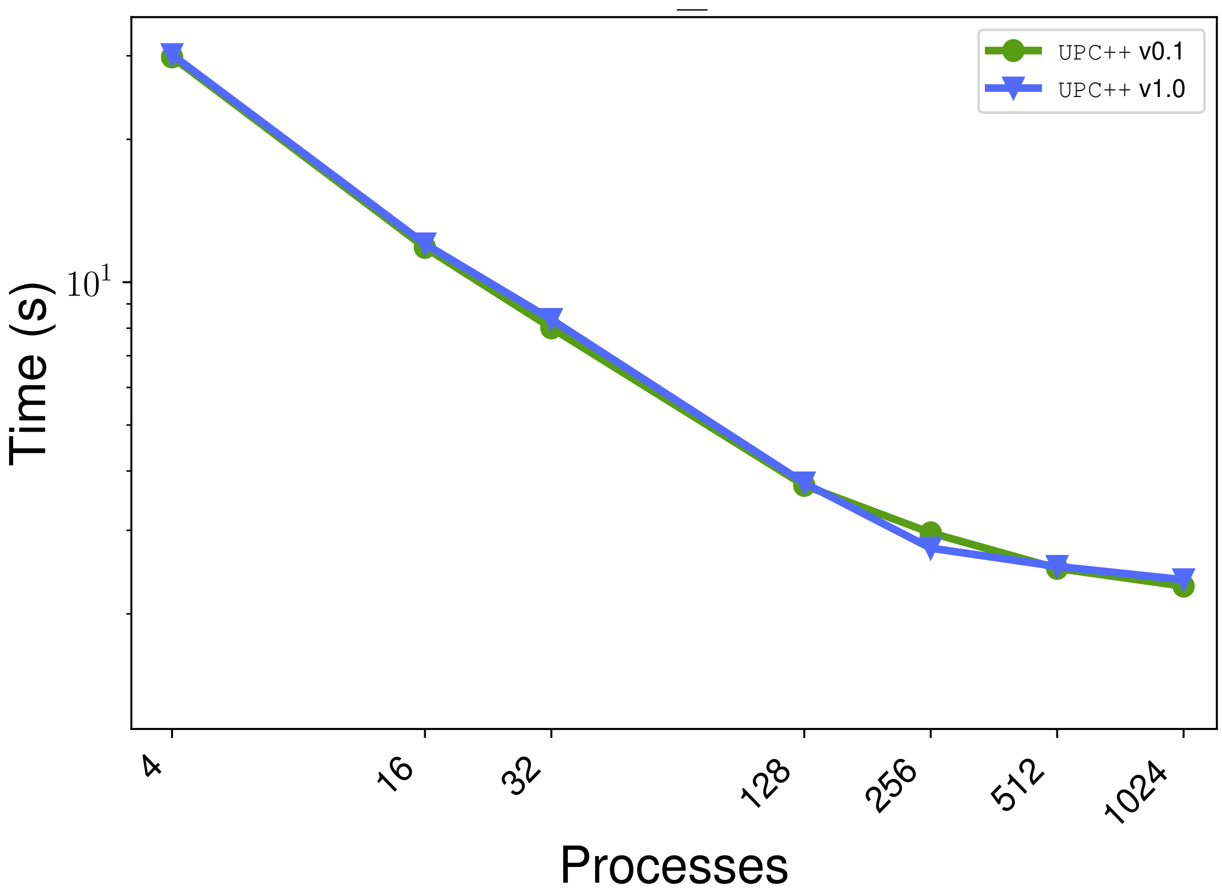
- UPC, Fortran 2008 coarrays, OpenSHMEM, Titanium
- Fork-join model: X10, Chapel
- DASH / DART (over MPI-3 RMA backend)
- Coarray C++

- Task-based models: HPX, Phalanx, Charm++, HabaneroUPC++

Differences with legacy UPC++ v0.1

- **Both implement PGAS model**
- Different APIs:
  - Current version avoids:
    - Implicit communication
    - Non-scalable data structures
  - Current version based on futures/promises (similar to C++11)
  - Leg. version uses *async/finish* syntax (like X10,Habanero-C)
- New functionalities:
  - *Futures* encapsulate values, *events* do not
  - Futures allow to attach callbacks
  - Easier to manage *future's* lifetime vs. *event*
  - *RPCs* can return a value, *asyncs* cannot

UPC++ v1.0 vs. v0.1 performance



- *SymPACK, supernodal solver for symmetric sparse matrices*
- *Implementation based on RPC & RMA*
- *Outperforms state-of-the-art solvers implemented using MPI*

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