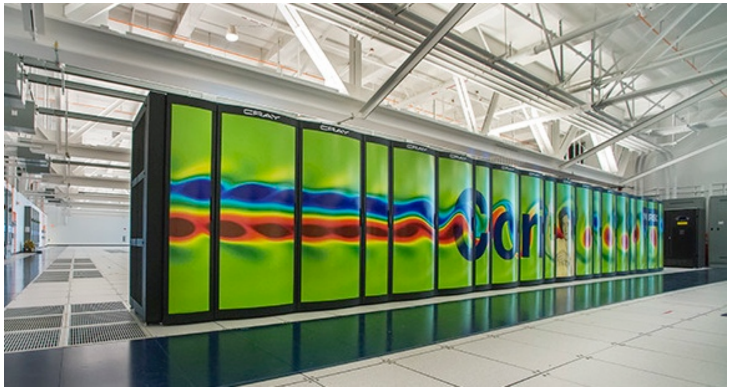
UPC++: Asynchrony and Active Messages

John Bachan
Lawrence Berkeley National Lab

UPC++

https://bitbucket.org/upcxx/upcxx/wiki/Home

- C++11 communication library for HPC.
- Similar to MPI:
  - One executable, parallel communicating instances.
  - Same scalability scope.
  - Semantics better matches underlying hardware.
- Semantic core:
  - Active Messages = one-sided messaging
  - PGAS = global read/write access.
  - Highly asynchronous API.
- UPC++ v1.0 is still in the works!
  - 5 years of strong research.
  - Continues to be extended.



This Talk

- Focus on our lowest level:
  - Concurrency with futures.
  - Active messages.
    - **Applications should use these!**
  - PGAS operations.
- Possible high-level features.
  - Killer features of other runtimes that we can do as a library.
- **All** presented API's are rough / not released.

Roadmap

- Concurrency
- Active Messages
- PGAS
- Big Extensions

UPC++ Futures

- `upcxx::future != std::future`
- Manages concurrency within thread, not across threads.
  - Just callbacks done better.
- Not thread-safe = *faster!*
  - `std::future` implementations tend to use atomics/locks.
  - Penalizes fine-grained futures.
  - We only incur a virtual function call.

UPC++ Future API

```
namespace upcxx {

template<typename ...T>
class future<T...>; // commonly single-valued: future<T>
```

UPC++ Future API

```
namespace upcxx {

template<typename ...T>
class future<T...>; // commonly single-valued: future<T>

// build trivially ready future
template<typename ...T>
future<T...> future_result(T&&...result);
```

UPC++ Future API

```
namespace upcxx {

template<typename ...T>
class future<T...>; // commonly single-valued: future<T>

// build trivially ready future
template<typename ...T>
future<T...> future_result(T&&...result);

// future that waits on all given futures, concatenates all values
// into one argument list
template<typename ...Futures>
future</*concat'd list*/> future_all(Futures ...many);
```

UPC++ Future API

```
namespace upcxx {

template<typename ...T>
class future<T...>; // commonly single-valued: future<T>

// build trivially ready future
template<typename ...T>
future<T...> future_result(T&&...result);

// future that waits on all given futures, concatenates all values
// into one argument list
template<typename ...Futures>
future</*concat'd list*/> future_all(Futures ...many);

// wait for result of "a",
// run continuation with "a"'s value(s),
// continuation can return a result or next future to wait on:
template<typename ...T, typename Lam>
future</*lam return type*/> operator>>(future<T...> a, Lam &&lam);
```

UPC++ Future API

```
namespace upcxx {

template<typename ...T>
class future<T...>; // commonly single-valued: future<T>

// build trivially ready future
template<typename ...T>
future<T...> future_result(T&&...result);

// future that waits on all given futures, concatenates all values
// into one argument list
template<typename ...Futures>
future</*concat'd list*/> future_all(Futures ...many);

// wait for result of "a",
// run continuation with "a"'s value(s),
// continuation can return a result or next future to wait on:
template<typename ...T, typename Lam>
future</*lam return type*/> operator>>(future<T...> a, Lam &&lam);

// make progress on all things until future completes,
// returns result
template<typename T>
T wait(future<T> fu);
void wait(future<> fu);
```

future's in Use

```
future<int> num = /*...*/;

future<double> scalar = /*...*/;

future<> buf_sent = upcxx::remote_put(
  dst_rank, dst_addr, buf, buf_size
);

future<int,double> all_done = future_all(
  num, scalar, buf_sent
);

future<> all_done_and_we_said_so =
  all_done >> [](int num, double scalar) {
    std::cout << "got num="<<num<<'\n';
    std::cout << "got scalar="<<scalar<<'\n';
    std::cout << "reclaiming sent buffer\n";
    delete[] buf;
  };
```

Roadmap

- Concurrency
- Active Messages
- PGAS
- Big Extensions

Active Messages

<u>Active Message</u> = asynchronous remote function call.

```
// returns immediately
void upcxx::send(
  intrank_t rank,
  upcxx::function<void()> &&am);

// execute any received active messages
void upcxx::progress();
```

Message functions may not:

- Call `upcxx::progress()`
- Block for communication.

Active Message Signature

```
(upcxx::function<void()> am)
```

Messages are executed at recipient, but:

- Take no arguments.
- Return nothing.
- Can’t access stack of recipient thread.

So…

- Must produce its effect using only recipient’s global variables.

C++ global variables = rank-local state

Easy Distributed Hashtable

```
// global variable: rank-local partition of “big” table
std::unordered_map<int,int> local_table_part;

// static assignment of keys to ranks
template<class T> intrank_t owner_of(T key)
  { return std::hash<T>()(key) % upcxx::global_ranks(); }

// associate key to val in big table
void dht_insert(int key, int val) {
  // go to rank that owns key
  upcxx::send(owner_of(key),
    [=]() {
      // on owner rank
      // “key” from outer scope available thanks to capture [=]
      // add key=val to local table
      local_table_part[key] = val;
    }
  );
}
```

Serializing Lambda’s

- Not “officially” doable as of C++14.
  - But works on many compilers.
- The default serializer for **all types** just copies their bytes.
- Specialize `upcxx::serialization<T>` for non-byte-serializable types.
  - We try to make this easy.
  - Forget this and seg-fault if you’re lucky!
- `upcxx::bind` to build lambdas containing serialization-specialized values.

Better Hashtable

```
// global variable: rank-local partition of "big" table
std::unordered_map<int,int> local_table_part;

// static assignment of keys to ranks
template<class T> intrank_t owner_of(T key)
  { return std::hash<T>()(key) % upcxx::global_ranks(); }

// apply f to value associated with key on whatever rank owns it
void dht_visit(int key, upcxx::function<void(int&)> f) {
  // go to rank that owns key
  upcxx::send(owner_of(key),
    std::move(f), // binds f into lambda
    [=](upcxx::function<void(int&)> f) {
      // local_table_part on recipient, not from where we came
      f(local_table_part[key]);
    }
  );
}
```

Hashtable Usage

```
// populate table
for(int x=0; x < 100; x++)
  dht_insert(x, x*x);

// -- synchronize ----------------

// print and modify
for(int x=0; x < 100; x++)
  dht_visit(x,
    [=](int &val) {
      // cout from all different ranks
      std::cout << x << "=" << val << "\n";
      val += 1;
    }
  );
}
```

AMR Put Ghost Zone

```
void amr_put_zone_and_signal(
    std::array<int,3> block_ijk,
    int zone_num,
    ndslice<double,3> data
  ) {
  intrank_t owner = amr_owner_of(block_ijk);

  // no need to check for self-send (owner == this rank)
  upcxx::send(owner,
    data, // requires special serialization
    [=](ndslice<double,3> data) {
      // find local storage for block_ijk
      // for (i,j,k) in zone:
      //   copy cell from data to local storage

      // signal zone's arrival, good options:
      // 1. decrement rank specific counter
      // 2. decrement block specific counter
    }
  );
}
```

AMR: Ghost Zone Consumer

```
// signal scheme #1: rank-local counter
//   amr_put_zone_and_signal decrements this.
int zones_missing; // global / rank-local

void amr_some_stencil_op() {
  // send out all ghoze zones
  for(auto &block: local_blocks)
    for(int zone: /* 0 ... 26 */)
      amr_put_zone_and_signal(...);

  zones_missing = 26 * local_blocks.size();

  while(zones_missing != 0)
    // amr_put_zone_and_signal lambdas are actively
    // decrementing zones_missing
    upcxx::progress();

  for(auto &block: local_blocks)
    /* do big compute for each block*/;
}
```

Simpler Than Two-Sided

UPC++ sender

- Lambda captures data to send.
- Lambda installs it remotely.

UPC++ receiver

- Spin on `upcxx::progress()` as needed.

MPI sender

- Generate tag-numbering scheme.
- Create send buffer.
- Generate tag number.
- `MPI_Isend`.

MPI receiver

- Generate tag-numbering scheme.
- Allocate receive buffers.
- Post all `MPI_Irecv`.
- Spin on `MPI_Testany`.
  - Decipher "what" from tag.
  - Install the data.

Two-Sided Restrictions

- Need to know who will be messaging you.
  - Very unnatural in some algorithms.
  - Might impose extra communication.
- Want a request/reply model.
  - Every rank acts as a “server” guarding their local data.
  - Other ranks ask for data, you send replies on-demand.
- Two-sided makes this cumbersome.
- AM’s make it easy!

AM Request/Reply

```
template<typename T>
upcxx::future<T> upcxx::remote_apply(
  intrank_t rank,
  upcxx::function<T()> request);

template<typename T>
upcxx::future<T> upcxx::remote_apply(
  intrank_t rank,
  upcxx::function<future<T>()> request);

// implementation:
// 1. upcxx::send function to rank
// 2. rank calls function, gets return value
// 3. if future, waits for it there
// 4. upcxx::send value back, trigger user's future
```

Request-Based Task

```
future<ndslice<double,2>> compute_thing(block_id id) {
  // 1. determine dependency
  block_id nbr_id = id.neighbor();
  intrank_t nbr_owner = owner_of(nbr_id); // dependency's owner

  // 2. go get my dependency
  future<ndslice<double,2>> nbr_data_fu =
    upcxx::remote_apply(nbr_owner,
      // lambda runs on owner of our dependency
      [=]() {
        return /*lookup data slice for nbr_id*/;
      }
    );

  // 3. do our compute once dependency arrives
  future<ndslice<double,2>> result =
    nbr_data_fu >>
    [=](ndslice<double,2> nbr_data) {
      return matmul(/*local data*/, nbr_data);
    };

  return result; // we return immediately
}
```

Same, But With Style

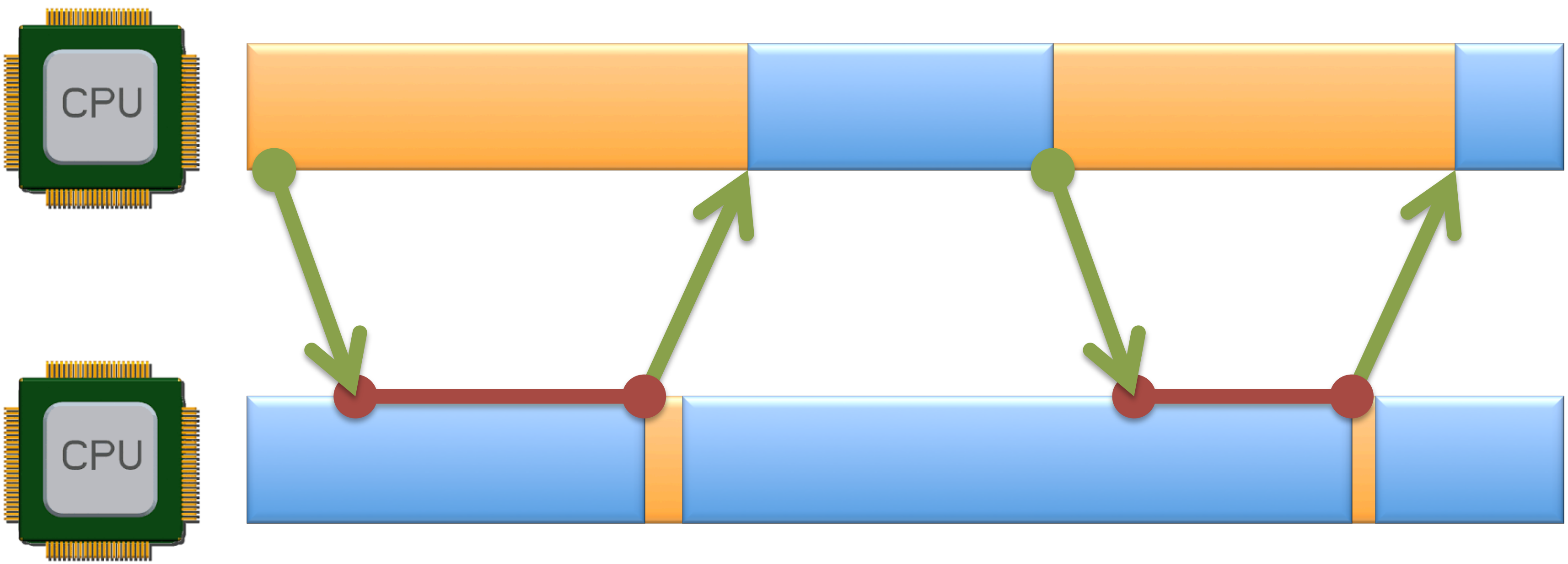
```
future<ndslice<double,2>> compute_thing(block_id id) {
  block_id nbr_id = id.neighbor();
  intrank_t nbr_owner = owner_of(nbr_id);

  // we return immediately
  return upcxx::remote_apply(nbr_owner,
      [=]() {
        return /*lookup data for nbr_id*/;
      }
    ) >>
    [=](ndslice<double,2> nbr_data) {
      return matmul(/*local data*/, nbr_data);
    };
}
```

Request/Reply Performance Issue

- Received lambda's are only executed cooperatively by calling `upcxx::progress`.
- <u>Attentiveness</u>: Latency between lambda arrival and processing.
  - Dictated by frequency application enters `progress()`.
  - A.k.a. average task length.

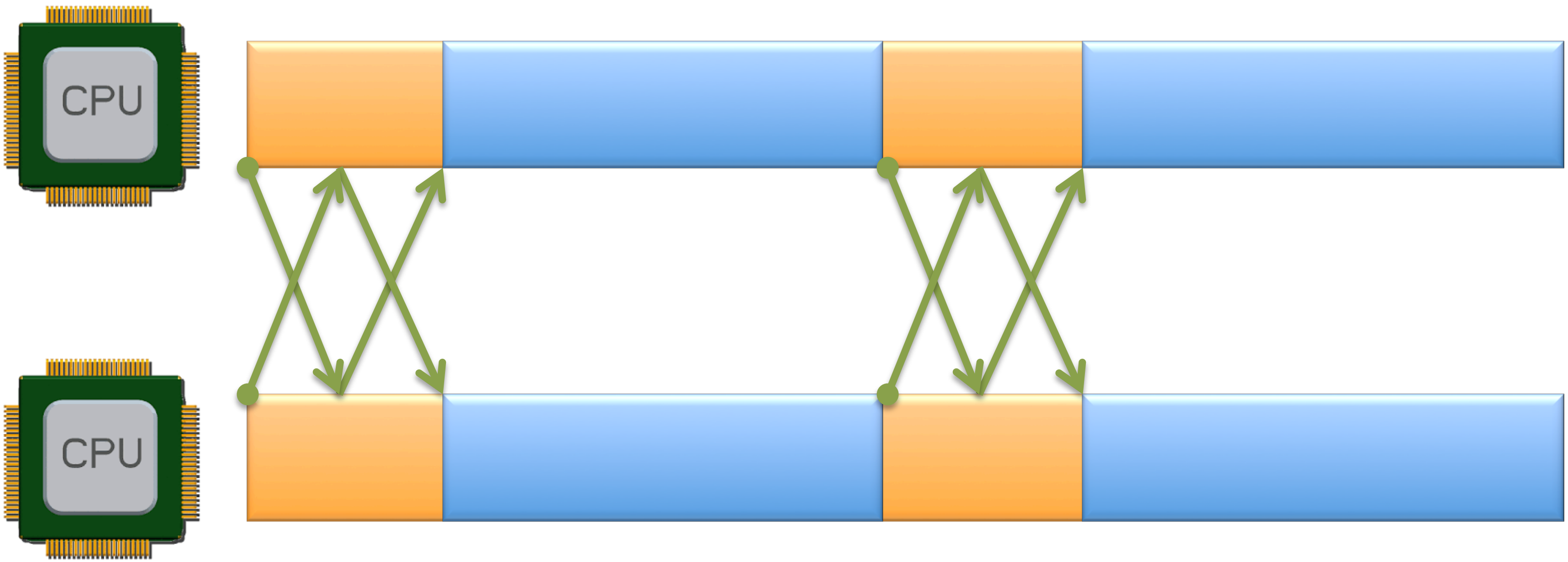
Attentiveness Nightmare



Task

```
while(<no tasks ready>)
  upcxx::progress();
```

Attentiveness Remedy #1: Aligned



Task

```
while(<expecting requests>)
  upcxx::progress();
```

Attentiveness Remedy #1: Aligned

- Globally align requesting phase:
  - All requests happen together while everyone is attentive.
  - Don’t enter tasks until all replies sent.
  - Requires good load-balancing!
- Synchronization options:
  - Point-to-point (2-sided, like MPI):
    - Must know requesters.
    - Precompute expected number of incoming requests.
      - Each request decrements counter on server.
    - Issue outgoing requests, wait for incoming replies.
    - Wait for request-decremented counter == 0.
    - Do work.
  - Global (not easy in MPI):
    - Unknown requesters.
    - Issue requests, wait for all replies.
    - Barrier (the price of the unknown).
    - Do work.

Attentiveness Remedy #2: Eager Send

Prefer push (unsolicited reply) over pull (request/reply).

- Must know requesters.
- Send replies eagerly as soon as data is computed.
- Requesters get data as soon as it exists. Impossible to do better.
- In MPI speak: space out `Isend` and `Irecv` to the max.
- Fast producers might hog slow consumers' heaps:
  - Consumer `malloc/operator new` fails. **YOU'RE DEAD**.
  - Solution #1: flow control protocol
  - Solution #2: rendezvous protocol
  - MPI has to internally duplicate your data to handle this. Many MPI's just block instead, causing deadlock.
  - We would "like" to present a non-buffering API alternative.

Attentiveness Remedy #3: $$$

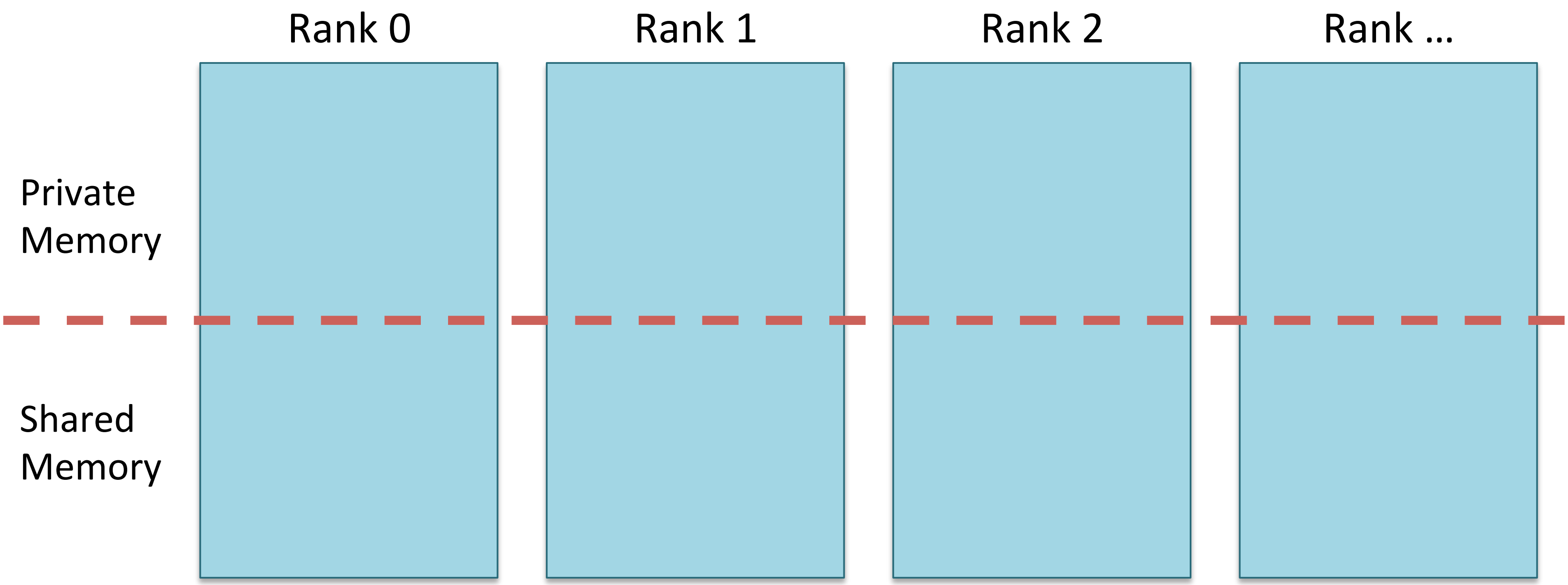
Dedicate a core/hyperthread to spinning on `upcxx::progress()`.

- Lose potential flops, but exascale has *cores to spare*.
- Legion-like:
  - Master thread maintains application state and handles AM's.
  - Offloads tasks to pool of worker threads.
  - Master only issues work that is hazard-free w.r.t. still pending work.
- Not Legion-like:
  - AM's and tasks all happen concurrently in thread pool.
  - Requires thread safe code.

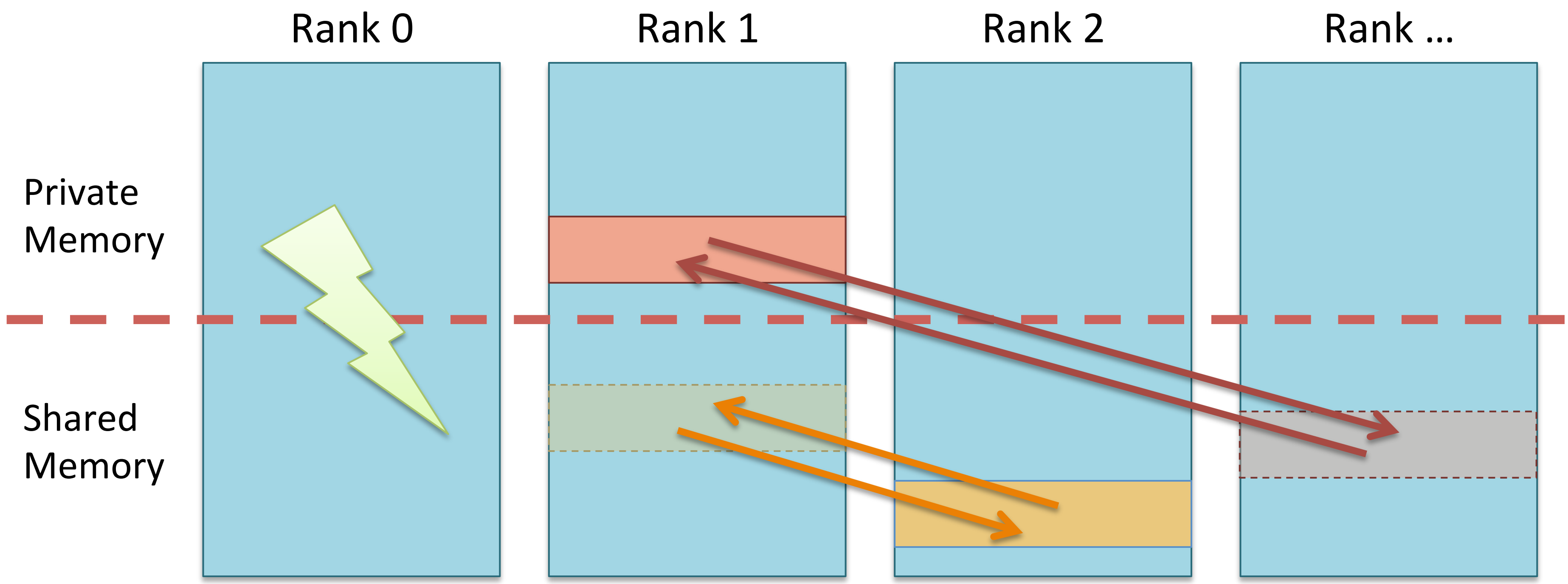
Attentiveness Remedy #4: PGAS

- Concurrency
- Active Messages
- PGAS
- Big Extensions

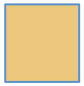
PGAS Memory Model



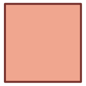
PGAS Memory Model



Native load/store.
Fastest.



Offloaded RDMA.
<u>No attentiveness required</u>.
Less fast.



CPU-assisted RDMA.
Attentiveness required.
Least fast.

Remote Memory Access

Contiguous **Put/Get** (NIC RDMA's):

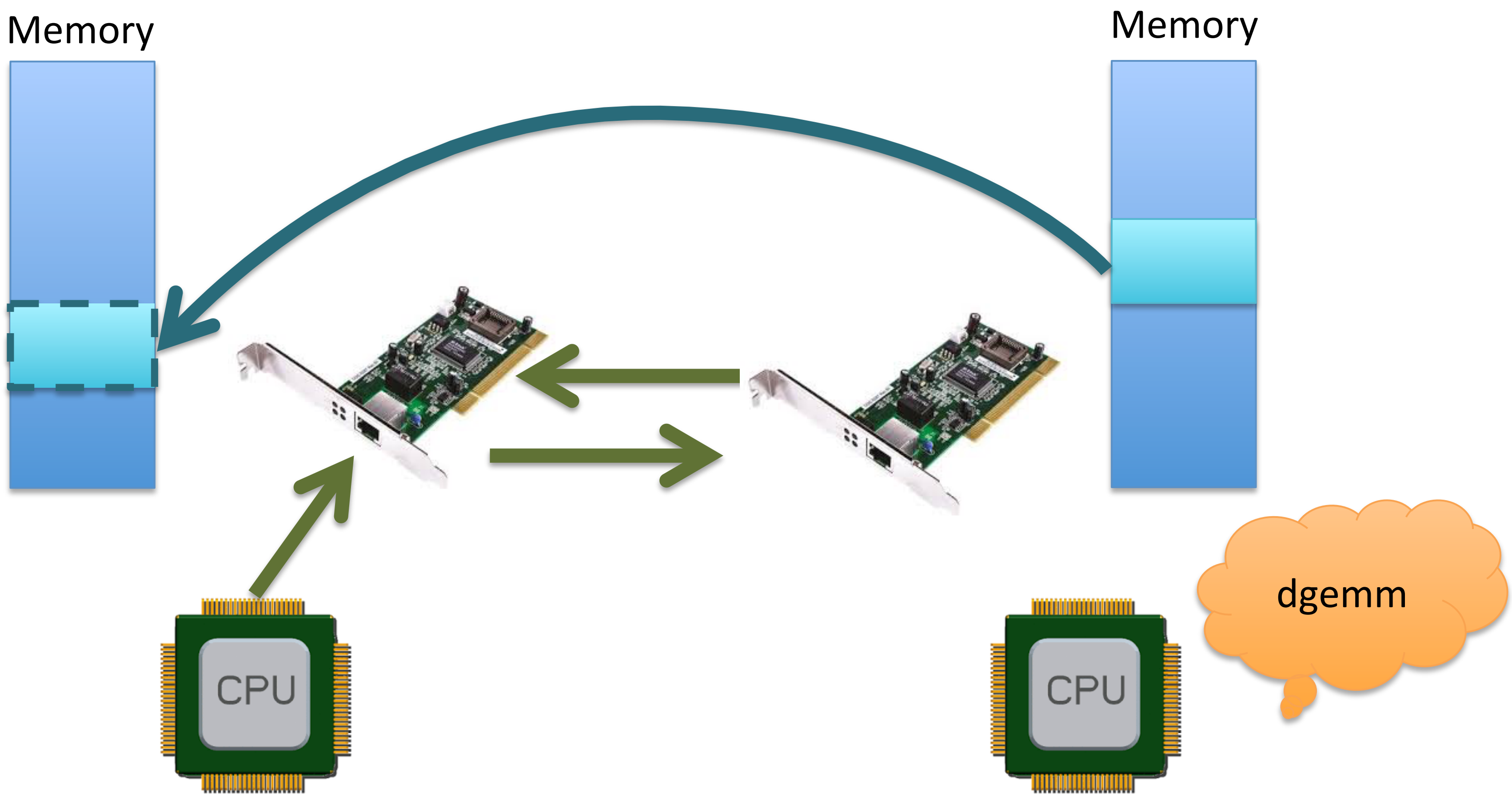
```
// for trivial T
template<typename T>
upcxx::future<> upcxx::remote_put(
  intrank_t d_rank, T *d_addr,
  T const *s_addr,
  size_t n);

template<typename T>
upcxx::future<> upcxx::remote_get(
  T *d_addr,
  intrank_t s_rank, T const *s_addr,
  size_t n);
```

“Consistency” Model

- Put completion = write is guaranteed visible to other getters.
- Get completion = your receiving buffer is filled.
- Concurrent Put+{Get|Put} to same memory is **undefined result.**
- Completion notification is per individual operation.
  - No fences.
  - Completion order is non-deterministic.
- Latency on Cray Aries > 1 microsecond.
  - Fine grained access is costly.

PGAS Get As Request



Attentiveness Remedy #4: PGAS

Back to attentiveness…

- Solution: use PGAS `remote_get`'s instead of AM requests.

Requirements:

- Get's per request should be small (best=1).
  - Implies contiguity of storage w.r.t. anticipated requests.
- Address of data must be available before request.
  - Allocate, compute, then broadcast addresses.

Good example:

- Block-sparse matrix, requests = whole blocks.
- Each block stored contiguously.

PGAS vs. AM's

- AM's are more productive (<u>my opinion</u>).
  - They do two-sided cleanly.
  - They do one-sided cleanly, MPI can't.
    - Attentiveness can be cured with dedicated core.
    - Attentiveness might not be your issue.
- PGAS performance advantages:
  - `put/get's` give highest guarantee of "zero-copy" data transfer.
    - Minimal intermediate buffering by network driver, OS, and NIC on both sender and receiver side.
  - No attentiveness required.
- PGAS disadvantages:
  - Require contiguity under all possible requests.
  - Locations must be setup ahead of time and addresses shared.
    - More code.
    - Extra synchronization/signalling.

Roadmap

- Concurrency
- Active Messages
- PGAS
- Big Extensions

Lofty Goal

- X10, Charm++, HPX, Legion all built with… *Active messages!!!*
- For each "cool" feature of runtime X:
  - Implement as UPC++ library code.
  - Let users opt-in.
  - Defeat the tyranny of all-or-nothing runtimes.

Quiescence Detection

Quiescence:

- All ranks out of “work”.
- No messages in flight which could create work.

Quiescence Detection: Hard to implement!

- Killer feature of X10 and Charm++.

Can implement as library on UPC++.

Quiescence Detection API

```
// same interface as upcxx::send,
// adds additional tracking
void qd::send(intrank_t rank, function<void()> &&am);

// returns true when we're globally quiescent.
bool qd::progress(bool locally_quiescent);
```

Quiescence Guts

```
uint64_t qd::_send_n = 0; // per-rank state as globals
uint64_t qd::_recv_n = 0;

void qd::send(intrank_t rank, function<void()> &&am) {
  qd::_send_n += 1; // send-side bookkeeping
  upcxx::send(rank)(
    std::move(am),
    [](function<void()> &am) {
      qd::_recv_n += 1; // receive-side bookkeeping
      am();
    }
  );
}

bool qd::progress(bool locally_quiescent) {
  // ~160 lines of tricky code...
  // wraps upcxx::progress()
}
```

Unbalanced Tree Search

```
// rank-local node list
std::deque<uts_node_t> local_nodes;

void do_uts() {
  while(!qd::progress(local_nodes.empty()) {
    uts_node_t popped = /*pop from local_nodes*/;

    for(/*each child of popped*/) {
      intrank_t rank = /*hash(child)*/;

      qd::send(rank, [=]() {
        local_nodes.push_front(child);
      });
    }
  }
  // all queues empty, no messages in flight
}
```

HPX À La Carte

- AGAS: distributed directory for transiently moving objects.
  - One of HPX's *killer features.*
- UPC++ sketch:
  - Rough implementation < 1000 lines of code

```
template<typename Key, typename Val>
void agas::send(Key const &key, function<void(Val&)> &&am);

template<typename Key>
void agas::relocate(Key const &key, intrank_t rank);

template<typename Key>
void agas::erase(Key const &key);
```

AGAS Example

```
typedef std::tuple<int,int> block_key;
typedef ndslice<double,2> block_data;

// on rank 1
block_data stuff = /*...*/;

agas::send<block_key, block_data>(
  block_key{0,0},
  [=](block_data &d) {
    dgemm(d, d, stuff); // mul stuff onto d
  }
);

// on rank 2
agas::relocate(block_key{0,0}, 99);
```

Composing Extensions

- Q: `agas` within a quiescence context?
- A: parameterize `upcxx::send` out of `agas`.

```
typedef void
  send_signature(intrank_t, upcxx::function<void()>&&);

// augment all of agas::* with "send" parameter
template<typename Key, typename Val>
void agas::send(Key const &key,
  upcxx::function<void(Val&)> &&am,
  upcxx::function<send_signature> &&send = upcxx::send);

// usage
while(!qd::progress(/*...*/)) {
  // ...
  agas::send<Key,Val>(key, [](Val&){/*...*/}, qd::send);
  // ...
}
```

…

Done

Put/Get Performance

- Overhead per put/get **much** higher than local load/store.
  - Latency on CRAY Aries > 1 microsecond.
- Blocking for completion can be costly.
- `future's` allow us to write <u>never blocking</u> code.

Coding for Concurrency

```
// indivually blocking
// = WORST
wait(remote_put(...));
wait(remote_put(...));
wait(remote_put(...));
```

Coding for Concurrency

```
// indivually blocking
// = WORST
wait(remote_put(...));
wait(remote_put(...));
wait(remote_put(...));

// batch blocking = BETTER
wait(future_all(
  remote_put(...),
  remote_put(...),
  remote_put(...)
));
```

Coding for Concurrency

```
// indivually blocking
// = WORST
wait(remote_put(...));
wait(remote_put(...));
wait(remote_put(...));

// batch blocking = BETTER
wait(future_all(
  remote_put(...),
  remote_put(...),
  remote_put(...)
));
```

```
// non-blocking = BEST!
future_all(
  remote_put(...),
  remote_put(...),
  remote_put(...)
) >>
[=]() {
  // all puts complete
};
```

Non-Blocking, Serial

```
// serialized via blocking
wait(remote_put(...));
wait(remote_put(...));
wait(remote_put(...));
```

Non-Blocking, Serial

```
// serialized via blocking
wait(remote_put(...));
wait(remote_put(...));
wait(remote_put(...));

// same consistency but non-blocking
remote_put(...) >>
[=]() {
  return remote_put(...) >>
    [=]() {
      return remote_put(...) >>
        [=]() {
          // all puts done
        };
    };
}
```

Non-Blocking Non-Concurrent

```
// serialized via blocking
wait(remote_put(...));
wait(remote_put(...));
wait(remote_put(...));

// same consistency but non-blocking
remote_put(...) >>
[=]() {
  return remote_put(...) >>
    [=]() {
      return remote_put(...) >>
        [=]() {
          // all puts done
        };
    };
}
```

No performance boost unless composed with concurrency elsewhere.

Composable Concurrency

Concurrency across software boundaries:

```
// in lib-foo
future<> foo_put() {
  return future_all( // put some buffers
    remote_put(...),
    remote_put(...),
    remote_put(...)
  );
}
```

Composable Concurrency

Concurrency across software boundaries:

```
// in lib-foo
future<> foo_put() {
  return future_all( // put some buffers
    remote_put(...),
    remote_put(...),
    remote_put(...)
  );
}
```

```
// in lib-bar
future<T*> bar_get() {
  return future_all( // get some buffers
      remote_get(...),
      remote_get(...)
    ) >>
    [=]() { // do some unpacking
      T *user_buf = new T[...];
      // fill user_buf from "get" buffers
      return user_buf;
    };
}
```

Composable Concurrency

Concurrency across software boundaries:

```
// in lib-foo
future<> foo_put() {
  return future_all( // put some buffers
    remote_put(...),
    remote_put(...),
    remote_put(...)
  );
}
```

```
// in lib-bar
future<T*> bar_get() {
  return future_all( // get some buffers
      remote_get(...),
      remote_get(...)
    ) >>
    [=]() { // do some unpacking
      T *user_buf = new T[...];
      // fill user_buf from "get" buffers
      return user_buf;
    };
}
```

```
// foo & bar in parallel
future_all(
  foo_put(),
  bar_get()
) >>
[=](T *bar_buf) {
  // foo_put complete
  // bar_get result available
};
```

Dynamic Concurrency

- `operator>>` allows given continuation to return another `future`.
- Allows recursion.
- Recursion is Turing-complete.

Therefor:

- ➢ `operator>>` can handle any control-flow, no matter how convoluted!
- ➢ Haskell'ers rejoice!

Dynamic Concurrency (example)

```
// traditional spinlock loop (most naive implementation)
void local_lock_acquire(std::atomic<int> *flag) {
  int expected;
  do {
    expected = 0;
    flag->compare_exchange_strong(expected, 1);
  } while(expected != 0);
}
```

Dynamic Concurrency (example)

```
// NIC supported primitive for agreeable types T
template<typename T>
future<T> upcxx::remote_compare_exchange(
  intrank_t rank, T *addr, T expected, T desired
);

// future-recursive spinlock loop
future<> remote_lock_acquire(intrank_t rank, int *flag) {
  return remote_compare_exchange(rank, flag, 0, 1) >>
    [=](int found) {
      if(found == 0) // we swapped, have the lock
        return future_result();
      else // try again
        return remote_lock_acquire(rank, addr);
    };
}
```

Dynamic Composability

```
// from previous slide
future<> remote_lock_acquire(intrank_t rank, int *flag);

// get two spin locks concurrently
future_all(
  remote_lock_acquire(some_rank1, some_addr1),
  remote_lock_acquire(some_rank2, some_addr2)
) >>
[=]() {
  // we have both locks!
};
```

Coolness: running two loops and their atomics concurrently!

Dynamic Composability

```
// from previous slide
future<> remote_lock_acquire(intrank_t rank, int *flag);

// get two spin locks concurrently
future_all(
  remote_lock_acquire(some_rank1, some_addr1),
  remote_lock_acquire(some_rank2, some_addr2)
) >>
[=]() {
  // we have both locks!
};
```

Coolness: running two loops and their atomics concurrently!

WARNING: Dumb code! Grabbing locks in parallel is a recipe for deadlock.