PGAS in C++:
A Portable Abstraction for Distributed Data Structures

BENJAMIN BROCK
About Me

- PhD candidate at Berkeley

- Advised by Kathy Yelick and Aydın Buluço

- Work on large-scale parallel systems

- Use a lot of LBL, ORNL supercomputers
This Talk

Background: how do we write a program for a supercomputer?

Introduce PGAS Model, RDMA

Building Remote Pointer Types

Building Distributed Data Structures

Extending to GPUs
This Talk

Background: how do we write a program for a supercomputer?

Introduce PGAS Model, RDMA

Building Remote Pointer Types

Building Distributed Data Structures

Extending to GPUs
This Talk

**Background**: how do we **write a program** for a **supercomputer**?

Introduce **PGAS Model**, **RDMA**

Building **Remote Pointer** Types

Building **Distributed Data Structures**

Extending to **GPUs**
This Talk

**Background**: how do we write a program for a supercomputer?

Introduce **PGAS Model**, RDMA

Building **Remote Pointer** Types

Building **Distributed Data Structures**

Extending to **GPUs**
This Talk

Background: how do we write a program for a supercomputer?

Introduce PGAS Model, RDMA

Building Remote Pointer Types

Building Distributed Data Structures

Extending to GPUs
What This Talk Is Not

- A distributed implementation of the STL

- A full evaluation of parallel computing models
What This Talk Is Not

- A distributed implementation of the STL

- A full evaluation of parallel computing models
Background: How to supercompute?
What is a Cluster?

- A collection of **nodes**, connected by a **network**.
How do I program one?

- **Message Passing** - processes issue matching **send** and **receive** calls

![Diagram showing message passing between Node 0 and Node 1]
How do I program one?

- **Message Passing** - processes issue matching `send` and `receive` calls

```cpp
// Calculate data
auto values = algorithm(1.0f, 3, data);

// Send data to proc. 1
MPI_Send(values.data(), values.size(), MPI_FLOAT, 1, 0, MPI_COMM_WORLD);

// Data is now sent.

// Allocate space for data
std::vector<float> recv_values(num_values);

// Receive data from proc. 0
MPI_Recv(recv_values.data(), num_values, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);

// Data is now in `recv_values`
```
How do I program one?

- **Message Passing** - processes issue matching `send` and `receive` calls.

```
// Calculate data
auto values = algorithm(1.0f, 3, data);

// Send data to proc. 1
MPI_Send(values.data(), values.size(), MPI_FLOAT, 1, 0, MPI_COMM_WORLD);

// Data is now sent.

// Allocate space for data
std::vector<float> recv_values(num_values);

// Receive data from proc. 0
MPI_Recv(recv_values.data(), num_values, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);

// Data is now in `recv_values`
```
How do I program one?

- **Message Passing** - processes issue matching `send` and `receive` calls.

```
Node 0
Send MSG(1)
```

```
Node 1
Receive MSG(0)
```

```c
// Calculate data
auto values = algorithm(1.0f, 3, data);

// Send data to proc. 1
MPI_Send(values.data(), values.size(), MPI_FLOAT, 1, 0, MPI_COMM_WORLD);

// Data is now sent.
```

```c
// Allocate space for data
std::vector<float> recv_values(num_values);

// Receive data from proc. 0
MPI_Recv(recv_values.data(), num_values, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);

// Data is now in `recv_values`
```

**P0 and P1 must both participate in message.**
How do I program one?

- **Message Passing** - processes issue matching `send` and `receive` calls

- **RDMA** - directly read/write to **remote memory**
How do I program one?

- **Message Passing** - processes issue matching send and receive calls

- **RDMA** - directly read/write to remote memory

```cpp
auto remote_ptr = ...;
// Calculate data
auto values = algorithm(1.0f, 3, data);

// Send data to proc. 1
BCL::memcpy(remote_ptr, values.data(), values.size() * sizeof(float));

BCL::flush();

// Data is copied.
```
How do I program one?

- **Message Passing** - processes issue matching `send` and `receive` calls

- **RDMA** - directly read/write to remote memory

```cpp
auto remote_ptr = ...;
// Calculate data
auto values = algorithm(1.0f, 3, data);
// Send data to proc. 1
BCL::memcpy(remote_ptr, values.data(), values.size() * sizeof(float));
BCL::flush();
// Data is copied.
```

**P1 does not participate in remote write.**
PGAS Model

- **Partitioned** - each process has its own shared segment

- **Global address space** - each proc’s shared segment can be referenced by other processes
PGAS Model

- **Partitioned** - each process has its own shared segment

- **Global address space** - each proc’s shared segment can be referenced by other processes
PGAS Model

- Partitioned - each process has its own shared segment

- Global address space - each proc’s shared segment can be referenced by other processes
Advantages of PGAS

- **Asynchronous** - RDMA operations executed by NIC

- Allows **irregular**, one-sided access

- Maps well to **data structure** ops
Advantages of PGAS

- Asynchronous - RDMA operations executed by NIC
- Allows irregular, one-sided access
- Maps well to data structure ops
Advantages of PGAS

- Asynchronous - RDMA operations executed by NIC
- Allows irregular, one-sided access
- Maps well to data structure ops
Advantages of PGAS

- Asynchronous - RDMA operations executed by NIC
- Allows irregular, one-sided access
- Maps well to data structure ops
Advantages of PGAS

- **Asynchronous** - RDMA operations executed by NIC

- Allows **irregular**, one-sided access

- Maps well to **data structure ops**
Advantages of PGAS

- Asynchronous - RDMA operations executed by NIC

- Allows irregular, one-sided access

- Maps well to data structure ops
Background: Parallel Programs
Parallel Programs

- **Multiple processes** are executing a program

- Each process has its own **memory space**

- Two methods of communication: **shared memory** and **message passing**
A SPMD Program

```cpp
#include <bcl/bcl.hpp>
#include <fmt/core.h>

int main(int argc, char** argv) {
    BCL::init();

    fmt::print("Hello from rank {}
        BCL::rank());

    BCL::finalize();
    return 0;
}
```
A SPMD Program

```cpp
#include <bcl/bcl.hpp>
#include <fmt/core.h>

int main(int argc, char** argv)
{
    BCL::init();
    fmt::print("Hello from rank {}", BCL::rank());
    BCL::finalize();
    return 0;
}
```

Output: `mpirun -n 4 ./test`

$ mpirun -n 4 ./test
Hello from rank 0
Hello from rank 1
Hello from rank 2
Hello from rank 3
A SPMD Program

```cpp
#include <bcl/bcl.hpp>
#include <fmt/core.h>

int main(int argc, char** argv)
{
    BCL::init();
    fmt::print(“Hello from rank {}”, BCL::rank());
    BCL::finalize();
    return 0;
}
```

Output: `mpirun -n 4 ./test`

```
$ mpirun -n 4 ./test
Hello from rank 0
Hello from rank 1
Hello from rank 2
Hello from rank 3
```

Each process runs the same program.
A SPMD Program

```cpp
#include <bcl/bcl.hpp>
#include <fmt/core.h>

int main(int argc, char** argv) {
    BCL::init();

    if (BCL::rank() == 2) {
        fmt::print("Rank 2 says hi!\n");
    }

    BCL::finalize();
    return 0;
}
```
A SPMD Program

```cpp
#include <bcl/bcl.hpp>
#include <fmt/core.h>

int main(int argc, char** argv) {
    BCL::init();
    if (BCL::rank() == 2) {
        fmt::print("Rank 2 says hi!
    }
    BCL::finalize();
    return 0;
}
```

Output: `mpirun -n 4 ./test`

$ mpirun -n 4 ./test
Rank 2 says hi!
A SPMD Program

```cpp
#include <bcl/bcl.hpp>
#include <fmt/core.h>

int main(int argc, char** argv)
{
    BCL::init();
    if (BCL::rank() == 2)
    {
        fmt::print("Rank 2 says hi!
        
        BCL::finalize();
    return 0;
}
```

Output: `mpirun -n 4 ./test`

```
$ mpirun -n 4 ./test
Rank 2 says hi!
```

Only one process runs portion in if statement.
Remote Pointers
Remote Pointers

- Remote pointers are **smart pointer classes** that may reference remote memory

1) We can use a remote pointer to do an **RDMA get / put**
2) Can also perform **atomic operations** (e.g. fetch-and-add, compare-and-swap)
3) If pointing to **shared memory in the current process**, can convert to regular (local) pointer
Building a Remote Pointer Type

template <typename T>
struct GlobalPtr {

... 

private:
    size_t rank_;  
    size_t offset_; 
};
Remote Pointer Types

```c++
template<typename T>
struct GlobalPtr {
    ...

private:
    size_t rank_;  
    size_t offset_;  
};

void memcpy(void* dest,
            GlobalPtr<void> src,
            size_t n) {
    // Issue remote get operation to  
    // copy `n` bytes from `src` to `dest`  
    backend::remote_get(dest, src, n, ...);
}
```
Remote Pointer Types

- Can build `memcpy` to support **reading/writing** from/to remote memory

- Can write `fetch_and_op`, `compare_and_swap`, etc. **atomic ops**

- Can **dereference** remote pointer
template <typename T>
struct GlobalPtr {

...

GlobalRef<T> operator*() {
    return GlobalRef<T>(*this);
}

private:
    size_t rank_;  
    size_t offset_;  
};
Remote Pointer Types

template<typename T>
struct GlobalPtr {

...<br><br>GlobalRef<T> operator*() {
    return GlobalRef<T>(*this);
}

private:<br>    size_t rank_;<br>    size_t offset_;<br>};

Not possible to return a regular T&, since memory may be remote.
Remote Pointer Types

```
template <typename T>
struct GlobalPtr {
    ...
    GlobalRef<T> operator*() const {
        return GlobalRef<T>(rank_, offset_);
    }

private:
    size_t rank_;  // rank
    size_t offset_; // offset
};

template <typename T>
struct GlobalRef {
    T & operator=(const T & value) {
        memcpy(ptr_, &value, sizeof(T));
        return value;
    }

    operator T() const {
        T value;
        memcpy(&value, ptr_, sizeof(T));
        return *static_cast<T*>(value);
    }

private:
    GlobalPtr<T> ptr_;  // pointer
};
```
Remote Pointer Types

- Allow referencing memory on another process

- Can support memcpy, atomics, pointer arithmetic, etc.

- Can support dereferencing, but must have custom reference type (cannot use T& across nodes)

- Limited to trivially copyable types
BCL Global Pointer Example

```cpp
BCL::GlobalPtr<int> ptr = nullptr;

if (BCL::rank() == 0) {
  ptr = BCL::alloc<int>(BCL::nprocs());
}

ptr = BCL::broadcast(ptr, 0);

ptr[BCL::rank()] = BCL::rank();
```
BCL Global Pointer Example

BCL::GlobalPtr<int> ptr = nullptr;

if (BCL::rank() == 0) {
    ptr = BCL::alloc<int>(BCL::nprocs());
}

ptr = BCL::broadcast(ptr, 0);

ptr[BCL::rank()] = BCL::rank();
BCL Global Pointer Example

# BCL Global Pointer Example

```cpp
BCL::GlobalPtr<int> ptr = nullptr;

if (BCL::rank() == 0) {
    ptr = BCL::alloc<int>(BCL::nprocs());
}

ptr = BCL::broadcast(ptr, 0);

ptr[BCL::rank()] = BCL::rank();
```

---

**Shared Segments**

- **R 0**: (0, 0xf) (Green)
- **R 1**: `nullptr`
- **R 2**: `nullptr`
- **R 3**: `nullptr`
BCL Global Pointer Example

BCL::GlobalPtr<int> ptr = nullptr;

if (BCL::rank() == 0) {
    ptr = BCL::alloc<int>(BCL::nprocs());
}

ptr = BCL::broadcast(ptr, 0);

ptr[BCL::rank()] = BCL::rank();
BCL::GlobalPtr<int> ptr = nullptr;

if (BCL::rank() == 0) {
    ptr = BCL::alloc<int>(BCL::nprocs());
}

ptr = BCL::broadcast(ptr, 0);

ptr[BCL::rank()] = BCL::rank();
BCL Global Pointer Example

```cpp
BCL::GlobalPtr<int> ptr = nullptr;
if (BCL::rank() == 0) {
    ptr = BCL::alloc<int>(BCL::nprocs());
}
ptr = BCL::broadcast(ptr, 0);
ptr[BCL::rank()] = BCL::rank();
```
BCL Global Pointer Example

BCL::GlobalPtr<int> ptr = nullptr;

if (BCL::rank() == 0) {
    ptr = BCL::alloc<int>(BCL::nprocs());
}

ptr = BCL::broadcast(ptr, 0);

ptr[BCL::rank()] = BCL::rank();
BCL Global Pointer Example

BCL::GlobalPtr<int> ptr = nullptr;

if (BCL::rank() == 0) {
    ptr = BCL::alloc<int>(BCL::nprocs());
}

ptr = BCL::broadcast(ptr, 0);

ptr[BCL::rank()] = BCL::rank();

Shared Segments

<table>
<thead>
<tr>
<th>R 0</th>
<th>R 1</th>
<th>R 2</th>
<th>R 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0xf)</td>
<td>(0, 0xf)</td>
<td>(0, 0xf)</td>
<td>(0, 0xf)</td>
</tr>
</tbody>
</table>
Data Structures

- Data structures are split into two types:

1) **Remote** data structures
   - Data located on a single process
   - Globally accessible

2) **Distributed** Data structures
   - Data distributed across many processes
   - Globally accessible
Data Structures

- Data structures are **split into two types:**

1) **Remote** data structures
   - Data *located* on a *single process*
   - *Globally* accessible

2) **Distributed** Data structures
   - Data *distributed* across many processes
   - *Globally* accessible
Data Structures

- Data structures are split into two types:

1) **Remote** data structures
   - Data located on a single process
   - Globally accessible

2) **Distributed** data structures
   - Data distributed across many processes
   - Globally accessible
Data Structures

- Data structures are split into two types:

  1) **Remote** data structures
     - Data located on a single process
     - **Globally** accessible

  2) **Distributed** Data structures
     - Data distributed across many processes
     - **Globally** accessible
Data Structures

- Data structures are **split into two types:**

1) **Remote** data structures
   - Data located on a **single process**
   - **Globally** accessible

2) **Distributed** Data structures
   - Data **distributed** across many processes
   - **Globally** accessible
Data Structures

- Data structures are **split into two types:**

1) **Remote** data structures
   - Data located on a single process
   - Globally accessible

2) **Distributed** data structures
   - Data distributed across many processes
   - Globally accessible

![Distributed Hash Table Diagram](image-url)
Data Structures

- **Constructors/destructors** that must be called **collectively**

- Each process has **global view** of data structure

- **Most methods** (e.g. insert, find) not collective

```cpp
#include <bcl/bcl.hpp>

int main(int argc, char **argv) {
    BCL::init();

    BCL::HashMap<std::string, int> map(BCL::nprocs());

    if (BCL::rank() == 0) {
        for (int i = 0; i < BCL::nprocs(); i++) {
            map.insert({std::to_string(i), i});
        }
    }
    ...
}
Data Structures

- **Constructors/destructors** that must be called **collectively**

- Each process has **global view** of data structure

- Most methods (e.g. insert, find) not collective

```cpp
#include <bcl/bcl.hpp>

int main(int argc, char **argv) {
  BCL::init();

  BCL::HashMap<std::string, int> map(BCL::nprocs());

  if (BCL::rank() == 0) {
    for (int i = 0; i < BCL::nprocs(); i++) {
      map.insert({std::to_string(i), i});
    }
  }

  ...
}
```

Each process invokes constructor collectively
Data Structures

- **Constructors/destructors** that must be called **collectively**

- Each process has **global view** of data structure

- **Most methods** (e.g. insert, find) not collective

```cpp
#include <bcl/bcl.hpp>

int main(int argc, char **argv) {
    BCL::init();

    BCL::HashMap<std::string, int> map(BCL::nprocs());

    if (BCL::rank() == 0) {
        for (int i = 0; i < BCL::nprocs(); i++) {
            map.insert({std::to_string(i), i});
        }
    }

    ...
}
```
Data Structures

- Constructors/destructors that must be called **collectively**
- Each process has **global view** of data structure
- Most methods (e.g. insert, find) not collective

```cpp
#include <bcl/bcl.hpp>

int main(int argc, char **argv) {
  BCL::init();
  BCL::HashMap<std::string, int> map(BCL::nprocs());
  if (BCL::rank() == 0) {
    for (int i = 0; i < BCL::nprocs(); i++) {
      map.insert({std::to_string(i), i});
    }
  }
  ...
}
```
Iteration - Global and Local

- “Global Iteration” supported over distributed range of elements

- “Local iteration” supported over local range of elements

```cpp
#include <bcl/bcl.hpp>

int main(int argc, char **argv) {
  BCL::init();

  BCL::HashMap<std::string, int> map = ...;

  if (BCL::rank() == 0) {
    for (auto iter = map.begin(); iter != map.end(); ++iter) {
      auto&& [key, value] = *iter;
      fmt::print("{}: {}", key, value);
    }
  }
  ...
}
```
Iteration - Global and Local

- “Global Iteration” supported over distributed range of elements

- “Local iteration” supported over local range of elements in process’ memory

```cpp
#include <bcl/bcl.hpp>

int main(int argc, char **argv) {
    BCL::init();
    BCL::HashMap<std::string, int> map = ...;
    if (BCL::rank() == 0) {
        for (auto iter = map.local_begin(); iter != map.local_end(); ++iter) {
            auto&& [key, value] = *iter;
            fmt::print("{}: {}", key, value);
        }
    }
    ...
}
```
Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, **directly read/write using RDMA.**
- **Resizing** must be done collectively

[1] ICPP’19, Brock, et. al
Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, 
  **directly read/write using RDMA.**
- **Resizing** must be done collectively

[1] ICPP’19, Brock, et. al
Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, **directly read/write using RDMA.**
- **Resizing** must be done collectively

![Diagram of insert operation](image)

[1] ICPP’19, Brock, et. al
Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, 
  directly read/write using RDMA.
- Resizing must be done collectively

[1] ICPP’19, Brock, et. al
Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, directly read/write using RDMA.
- **Resizing** must be done collectively

```
insert({k, v})
1) Calculate location
2) Request bucket (A_{FAO})
```

[1] ICPP’19, Brock, et. al
Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, \textbf{directly read/write using RDMA.}
- \textbf{Resizing} must be done collectively

\begin{itemize}
  \item \textbf{insert([k, v])}
  \item 1) Calculate location
  \item 2) Request bucket ($A_{FAO}$)
  \item 3) Insert item (W)
\end{itemize}

[1] ICPP’19, Brock, et. al
Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, 
  directly read/write using RDMA.
- Resizing must be done collectively

```
insert([k, v])
1) Calculate location
2) Request bucket (A_{FAO})
3) Insert item (W)
4) Mark bucket ready (A_{O})
```

[1] ICPP’19, Brock, et. al
Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, directly read/write using RDMA.
- Resizing must be done collectively

Best Case Cost: $A_{FAO} + W (+ A_O)$

**insert([k, v])**
1) Calculate location
2) Request bucket ($A_{FAO}$)
3) Insert item (W)
4) Mark bucket ready ($A_O$)

[1] ICPP’19, Brock, et. al
**Distributed Hash Table**

- Open addressing -- hash table buckets are split among processes
- To manipulate a bucket, directly read/write using RDMA.
- **Resizing** must be done collectively

---

**Best Case Cost:** 

\[ A_{FAO} + W (+ A_O) \]

---

Latency bound! Can we do better?

---

[1] ICPP’19, Brock, et. al
HashMapBuffer

- Constructed from a HashMap

- Similar to a range adaptor, but relaxes when operations take place

- Aggregates fine-grained insertions into large transfers

```cpp
#include <bcl/bcl.hpp>

int main(int argc, char **argv) {
    BCL::init();

    BCL::HashMap<br std::string, int> map = ...;
    BCL::HashMapBuffer<br std::string, int> buf(map);

    for (const auto& value : data) {
        buf.insert({value.key, value.value});
    }
    buf.flush();
    ...}
```
HashMapBuffer

- Constructed from a **HashMap**

- Similar to a **range adaptor**, but relaxes when operations take place

- **Aggregates** fine-grained insertions into large transfers

```cpp
#include <bcl/bcl.hpp>

int main(int argc, char **argv) {
    BCL::init();

    BCL::HashMap<std::string, int> map = ...;

    BCL::HashMapBuffer<std::string, int> buf(map);

    for (const auto& value : data) {
        buf.insert({value.key, value.value});
    }

    buf.flush();
    ...
}```
HashMapBuffer

- Constructed from a **HashMap**

- Similar to a **range adaptor**, but relaxes when operations take place

- **Aggregates** fine-grained insertions into large transfers

```cpp
#include <bcl/bcl.hpp>

int main(int argc, char **argv) {
    BCL::init();

    BCL::HashMap<std::string, int> map = ...;
    BCL::HashMapBuffer<std::string, int> buf(map);

    for (const auto& value : data) {
        buf.insert(std::pair{value.key, value.value});
    }
    buf.flush();
    ...
}
HashMapBuffer

- Constructed from a **HashMap**

- Similar to a **range adaptor**, but relaxes when operations take place

- **Aggregates** fine-grained insertions into large transfers

```cpp
#include <bcl/bcl.hpp>

int main(int argc, char **argv) {
  BCL::init();
  BCL::HashMap<std::string, int> map = ...;
  BCL::HashMapBuffer<std::string, int> buf(map);
  for (const auto& value : data) {
    buf.insert({value.key, value.value});
    buf.flush();
  }
  ...
}
```
Bulk Transfers Using Queues

Queues allow asynchronous all-to-all communication

Rank 0

Rank 1

Rank 2

Rank 3

[1] ICPP’19, Brock, et. al
Bulk Transfers Using Queues

Queues allow **asynchronous all-to-all communication**

[1] ICPP’19, Brock, et. al
Bulk Transfers Using Queues

Queues allow **asynchronous all-to-all communication**

![Diagram showing bulk transfers using queues across multiple ranks](image-url)
Bulk Transfers Using Queues

Queues allow **asynchronous all-to-all communication**

[1] ICPP’19, Brock, et. al
Genomics Benchmark

chr14, Cori Phase I

- PapyrusKV
- BCL without Buffer, GASNet-EX
- BCL with Buffer, GASNet-EX
- Meraculous
- perfect scaling

Runtime (s) vs Number of Nodes

[1] ICPP’19, Brock, et. al
Genomics Benchmark

chr14, Cori Phase I

3.7x Improvement with Aggregator

[1] ICPP’19, Brock, et. al
Genomics Benchmark

chr14, Cori Phase I

Match Perf. of Expert-Tuned Impl.
Comparison: Lines of Code

ISx Bucket Sort, Lines of Code

<table>
<thead>
<tr>
<th></th>
<th>BCL</th>
<th>Chapel</th>
<th>MPI</th>
<th>SHMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>72</td>
<td>244</td>
<td>838</td>
<td>899</td>
</tr>
</tbody>
</table>

Meraculous, Lines of Code

<table>
<thead>
<tr>
<th></th>
<th>BCL</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>600</td>
<td>4123</td>
</tr>
</tbody>
</table>
Some Data Structures We’ve Worked On

- Bloom Filters
- Queues
- Suffix Arrays
- Hash Tables
- Dense and Sparse Matrices
PGAS on GPUs
GPUs as a First-Class Computing Resource

- **GPUs** play an important role in modern large-scale computing systems

- **All three** DOE exascale systems will use **GPUs**

- ~10x more compute, BW

GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to **Infiniband** allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- Historically, network comm. was CPU-centric

1) Direct GPU access to Infiniband allows **GPU-to-GPU** network transfers

2) Fast in-node fabrics like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to Infiniband allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- Historically, network comm. was CPU-centric

1) Direct GPU access to Infiniband allows GPU-to-GPU network transfers

2) Fast in-node fabrics like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to Infiniband allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to Infiniband allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to Infiniband allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to **Infiniband** allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to **Infiniband** allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- Historically, network comm. was CPU-centric

1) Direct GPU access to Infiniband allows GPU-to-GPU network transfers

2) Fast in-node fabrics like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to Infiniband allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to **Infiniband** allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast **intra-node transfers**
GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to **Infiniband** allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- Historically, network comm. was CPU-centric

1) Direct GPU access to Infiniband allows GPU-to-GPU network transfers

2) Fast in-node fabrics like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- Historically, network comm. was CPU-centric

1) Direct GPU access to Infiniband allows GPU-to-GPU network transfers

2) Fast in-node fabrics like NVLink, Infinity Fabric allow very fast intra-node transfers
GPUs as a First-Class Computing Resource

- **Historically**, network comm. was CPU-centric

1) Direct GPU access to **Infiniband** allows **GPU-to-GPU** network transfers

2) **Fast in-node fabrics** like NVLink, Infinity Fabric allow very fast **intra-node transfers**
GPU Communication Libraries

- Communication libraries offering increasing support for GPU-to-GPU transfers

- Currently only PGAS-based libraries offer GPU-initiated communication

- NVSHMEM will utilize both GPUDirect RDMA and NVLink
GPU Communication Libraries

- Communication libraries offering increasing support for GPU-to-GPU transfers

- Currently only PGAS-based libraries offer GPU-initiated communication

- NVSHMEM will utilize both GPUDirect RDMA and NVLink
Remote Pointer Types

CPU Remote Pointer

BCL::GlobalPtr<int> ptr = nullptr;

if (BCL::rank() == 0) {
    ptr = BCL::alloc<int>(BCL::nprocs());
}

ptr = BCL::broadcast(ptr, 0);

ptr[BCL::rank()] = BCL::rank();
Remote Pointer Types

**CPU Remote Pointer**

```cpp
BCL::GlobalPtr<int> ptr = nullptr;
if (BCL::rank() == 0) {
    ptr = BCL::alloc<int>(BCL::nprocs());
}
ptr = BCL::broadcast(ptr, 0);
ptr[BCL::rank()] = BCL::rank();
```

**Remote GPU Pointer**

```cpp
BCL::cuda::ptr<int> ptr = nullptr;
if (BCL::rank() == 0) {
    ptr = BCL::cuda::alloc<int>(BCL::nprocs());
}
ptr = BCL::broadcast(ptr, 0);
ptr[BCL::rank()] = BCL::rank();
```
Remote Pointer Types

CPU Remote Pointer

BCL::GlobalPtr ptr = nullptr;
if (BCL::rank() == 0) {
    ptr = BCL::alloc<int>(BCL::nprocs());
}
ptr[BCL::rank()] = BCL::rank();

BCL::cuda::ptr<int> ptr = nullptr;
if (BCL::rank() == 0) {
    ptr = BCL::cuda::alloc<int>(BCL::nprocs());
}
ptr = BCL::broadcast(ptr, 0);
if (BCL::rank() == 1) {
    __global__ void kernel(BCL::cuda::ptr<int> ptr) {
        size_t tid = ...;
        ptr[tid] = tid;
    }
    kernel<<<1, BCL::nprocs()>>>(ptr);
}

Remote GPU Pointer (Accessing on GPU)
Remote Pointer Types for GPUs

CPU Remote Pointer

```
template <typename T>
struct GlobalPtr {

...

private:
    size_t rank_;
    size_t offset_;
};
```
Remote Pointer Types for GPUs

CPU Remote Pointer

template <typename T>
struct GlobalPtr {
  ...
private:
  size_t rank_;  
  size_t offset_;  
};

void memcpy(void* dest,  
            GlobalPtr<void> src,  
            size_t n) {  
  // Issue remote get operation to  
  // copy `n` bytes from `src` to `dest`  
  backend::remote_get(dest, src, n, ...);  
}
Remote Pointer Types for GPUs

GPU Remote Pointer

```
template<typename T>
struct ptr {

  ...

private:
  size_t rank_;  
  size_t offset_; 
};
```
Remote Pointer Types for GPUs

GPU Remote Pointer

```cpp
template <typename T>
struct ptr {
    ...

private:
    size_t rank_;  
    size_t offset_; 
};
```

```cpp
__host__ __device__
void memcpy(void* dest,
            cuda::ptr<void> src,
            size_t n) {
    // Issue remote get operation to 
    // copy `n` bytes from `src` to `dest`
    #ifdef __CUDA_ARCH__
    nvshmem_getmem(dest, 
                   src.rptr(), n, 
                   src.rank());
    #else
    ...
    #endif
}
Remote Pointer Types for GPUs

**GPU Remote Pointer**

```cpp
template <typename T>
struct ptr {
    // ...

    private:
    size_t rank_;  // rank and offset
    size_t offset_;  //...

};
```

On CPU, necessary to stage data if transferring to host (CPU) memory.
Distributed Data Structures on GPUs

- Recall that each process needs a table of pointers to access data.

- To implement GPU-side methods, need GPU-accessible table.

- Is this enough to implement GPU-side data structure methods?
Distributed Data Structures on GPUs

- Recall that **each process** needs a **table of pointers** to access data.

- To implement **GPU-side methods**, need **GPU-accessible** table.

- Is this enough to implement **GPU-side data structure methods**?
Distributed Data Structures on GPUs

- Recall that each process needs a table of pointers to access data.

- To implement GPU-side methods, need GPU-accessible table.

- Is this enough to implement GPU-side data structure methods?
Distributed Data Structures on GPUs

- Recall that each process needs a table of pointers to access data.
- To implement GPU-side methods, need GPU-accessible table.
- Is this enough to implement GPU-side data structure methods?
Passing Objects into CUDA Kernels

- Passing an **object by value** into a **CUDA kernel** results in a **copy**

- Object **likely destroyed** before kernel completes

- We need a **copy constructible** placeholder object

```cpp
__global__
void kernel(BCL::cuda::HashMap<int, int> map) {
    size_t tid = ...;

    size_t value = tid*2
    map[tid] = value;
}
...
BCL::cuda::HashMap<int, int> map(100);
kernel<<<1, 100>>>(map);
```
Passing Objects into CUDA Kernels

- Passing an **object by value** into a CUDA kernel results in a **copy**

- Object likely **destroyed** before kernel completes

- We need a **copy constructible** placeholder object

```cpp
__global__
void kernel(BCL::cuda::HashMap<int, int> map) {
    size_t tid = ...;
    size_t value = tid*2
    map[tid] = value;
}
...
BCL::cuda::HashMap<int, int> map(100);
kernel<<<1, 100>>>(map);
```
Passing Objects into CUDA Kernels

- Passing an **object by value** into a CUDA kernel results in a **copy**
- Object likely **destroyed** before kernel completes
- We need a **copy constructible** placeholder object

```cpp
__global__
void kernel(BCL::cuda::HashMap<int, int> map) {
    size_t tid = ...;
    size_t value = tid*2
    map[tid] = value;
}
...
BCL::cuda::HashMap<int, int> map(100);
kernel<<<1, 100>>>(map);
```
Passing Objects into CUDA Kernels

- Passing an **object by value** into a CUDA kernel results in a copy

- Object **likely destroyed** before kernel completes

- We need a **copy constructible** placeholder object

```
__global__
void kernel(BCL::cuda::HashMap<int, int> map) {
    size_t tid = ...
    size_t value = tid*2
    map[tid] = value;
}
...
BCL::cuda::HashMap<int, int> map(100);
kernel<<<1, 100>>>(map);
```
Passing Objects into CUDA Kernels

- Passing an **object by value** into a CUDA kernel results in a **copy**

- Object **likely destroyed** before kernel completes

- We need a **copy constructible** placeholder object

```cpp
__global__
void kernel(BCL::cuda::HashMap<int, int> map) {
    size_t tid = ...;
    size_t value = tid*2
    map[tid] = value;
}
...
BCL::cuda::HashMap<int, int> map(100);
kernel<<<1, 100>>>(map);
```
Passing Objects into CUDA Kernels

- Passing an **object by value** into a **CUDA kernel** results in a **copy**

- Object **likely destroyed** before kernel completes

- We need a **copy constructible** placeholder object

```cpp
__global__
void kernel(BCL::cuda::HashMap<int, int> map) {
    size_t tid = ...;

    size_t value = tid*2
    map[tid] = value;
}

...

BCL::cuda::HashMap<int, int> map(100);

kernel<<<1, 100>>>(map);
```
Using GPU Views within Kernels

- First create a `dmatrix_view` view object

- `dmatrix_view` has $O(1)$ copy constructor (does not copy data)

- View can be used to access data on GPU

```cpp
__global__
void kernel(cuda::dmatrix_view<float> x_view) {
    size_t tid = ...;

    size_t i = tid / x.shape()[0];
    size_t j = tid % x.shape()[0];

    x_view[{i, j}] = tid;
}
```

```cpp
cuda::DMatrix<float> a({8, 8});
kernelllll(1, 64)(cuda::dmatrix_view(a));
```
Wrap-Up

- **Remote pointer types** are a **useful abstraction** for implementing distributed data structures

- Extendable to **multi-GPU data structures** both intra-node and multi-node

- Having the correct **high-level distributed data structures** can unlock performance competitive with highly tuned implementations
Pointers

Links

BCL, Our PGAS-Based C++ Distributed Data Structures Library
https://github.com/berkeley-container-library/bcl

My Website
https://cs.berkeley.edu/~brock

Interested in irregular data structures? Check out my other talk:
GraphBLAS: Building a C++ Matrix API for Graph Algorithms (CppCon’21)