HERDS: Heterogeneous, Resilient, Distributed System for Key-Value Programming

Max Grossman, Matthew Whitlock, Vivek Sarkar
{max.grossman, mwhitlock9, vsarkar}@gatech.edu
Habanero Extreme Scale Software Research Laboratory

October 27 2021

Background related to key-value pairs

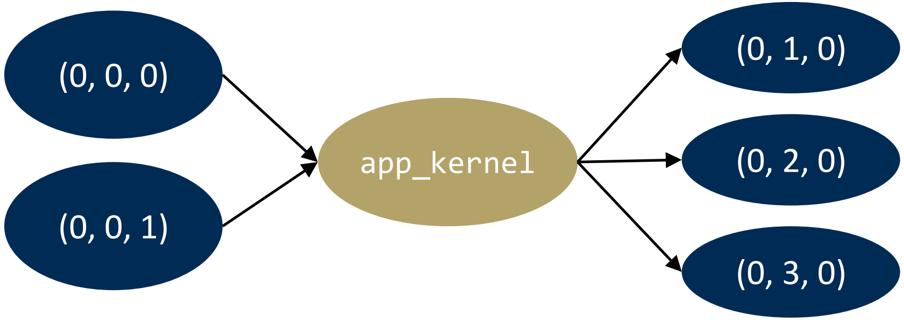
- CnC (Concurrent Collections)
  - Dataflow programming model that uses steps (computation), tags (control flow), and items (data) to define scalable, portable parallel programs
  - Focused on homogeneous SMP+cluster platforms, not an explicit key-value store but uses analogous concepts (tags, items) to coordinate execution.
- OCR (Open Community Runtime)
  - Event Driven Tasks as a basic unit of computation, Data Blocks for migratable data storage and resilience support
  - No built in support for heterogeneity or scalable parallelism
- Spark PairRDDs
  - Distributed, resilient arrays of key-value pairs
  - Ancestry tracking approach to resiliency (also used by HERDS)
  - No built in support for heterogeneity or scalable parallelism

HERDS Summary

A distributed key-value store with an integrated computational engine over a high performance network communication runtime.

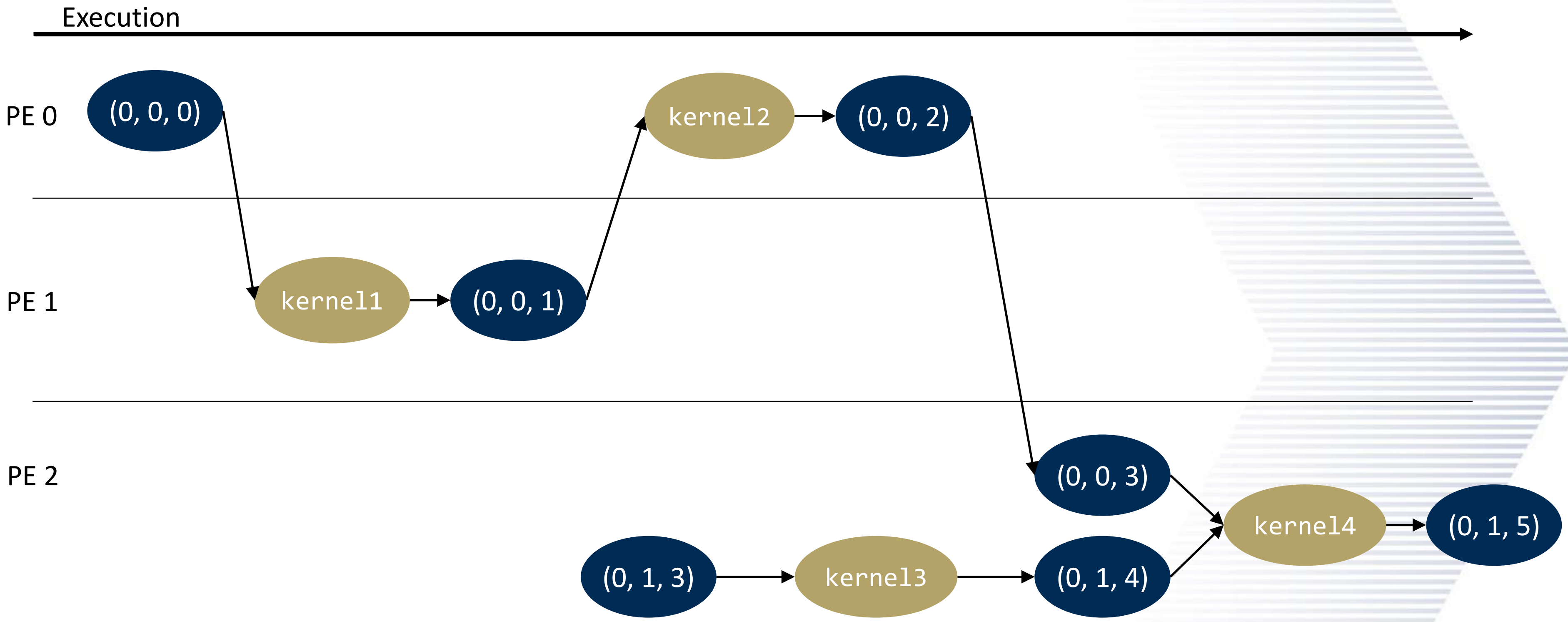
Programmer expresses their application as a dependency graph of key-value pairs, with application kernels that take N key-value pairs as input and produce M key-value pairs as output.

HEterogeneous: Can currently target CPUs or GPUs, Bluefield support in-progress
Resilient: Supports task replication/validation and replay
Distributed: Multi-node runtime, using the conveyor library for high performance communication



Motivating applications primarily from the realm of graph analysis and machine learning.

HERDS Summary



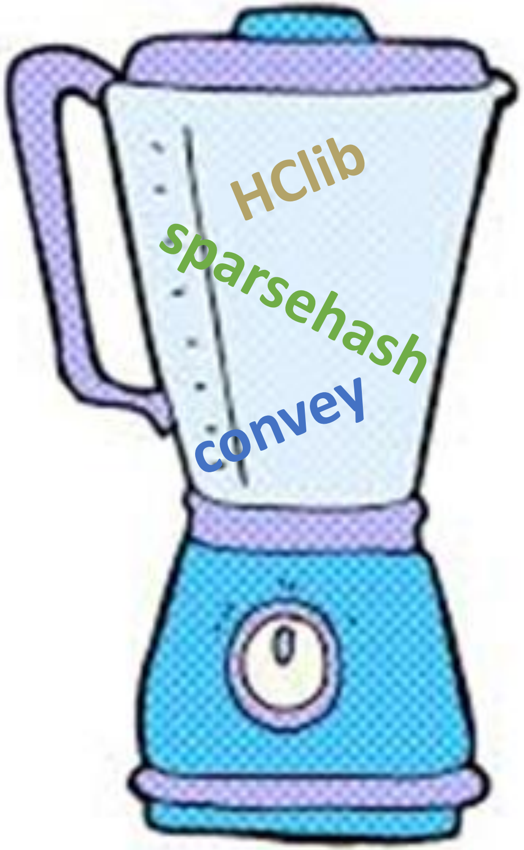
Computation graph is executed lazily, except for when data crosses PE boundaries

HERDS Summary

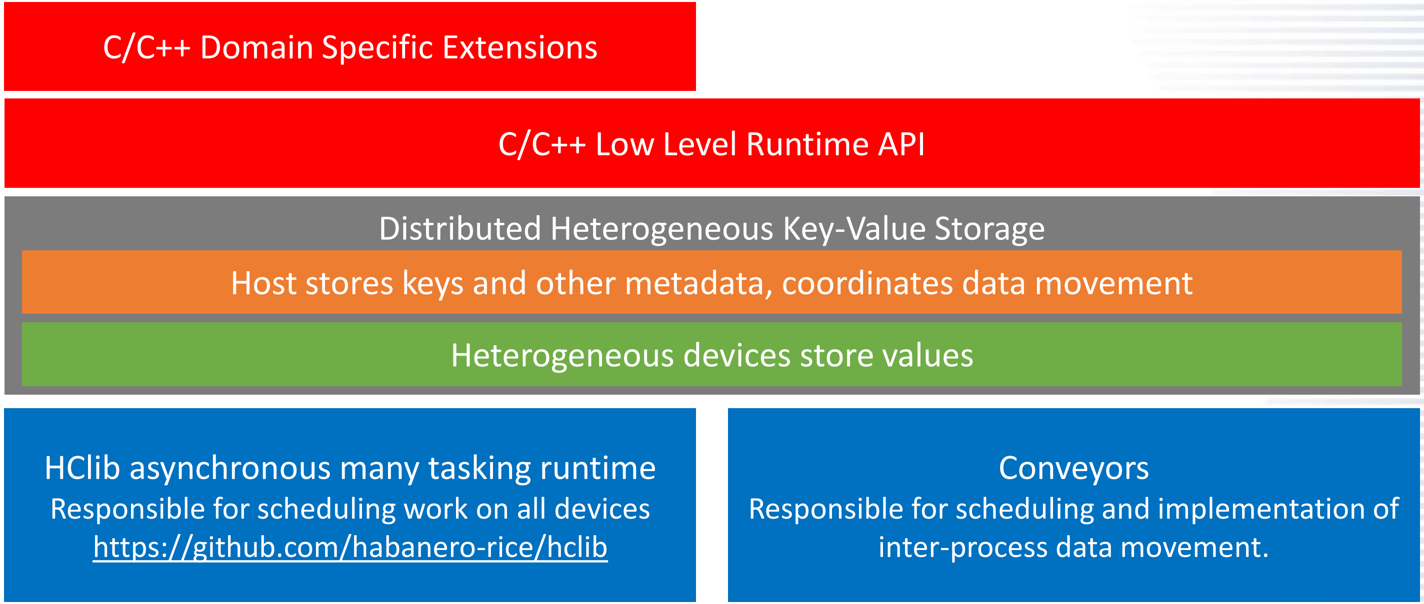
HClib = shared memory, asynchronous tasking

Conveyors = high throughput, distributed communication

Sparsehash = local, efficient hash map implementation



HERDS Software Stack



C/C++ Domain Specific Extensions

C/C++ Low Level Runtime API

Distributed Heterogeneous Key-Value Storage

Host stores keys and other metadata, coordinates data movement

Heterogeneous devices store values

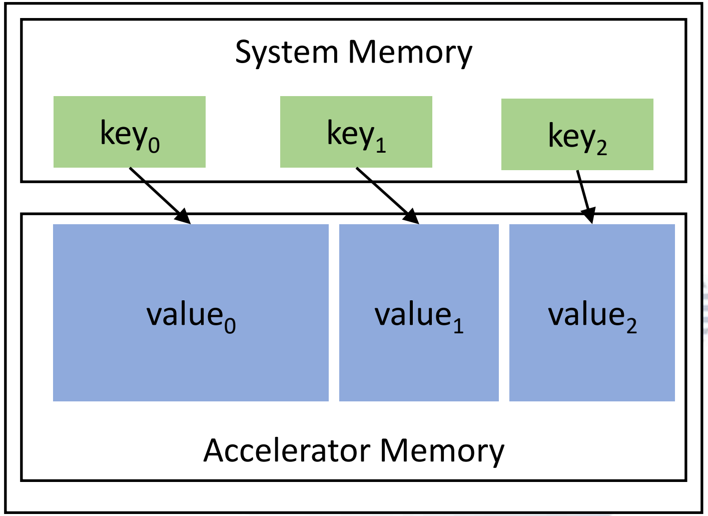
HClib asynchronous many tasking runtime
Responsible for scheduling work on all devices
https://github.com/habanero-rice/hclib

Conveyors
Responsible for scheduling and implementation of inter-process data movement.

HERDS runtime is concurrent but not parallel. HClib is only run with 1 worker thread, upon which all work is multiplexed. HERDS uses SHMEM for parallelism.

HERDS Data Model

- Core data structure: distributed hash table.
- Map from 3-tuple keys to void* values
  - Keys stored on memory owned by control processors
  - Values stored in any memory space
  - Management, control logic executed by control processors (e.g. find PE with key *X*)
  - Computationally heavy workloads executed on accelerators (e.g. map kernel across a range of keys)
- Key-value pairs are immutable but versioned
- Runtime supports void* values. Higher level programming models are responsible for offering higher level abstractions.



HERDS C/C++ Programming Model

Programmers write transforms (i.e. lambdas) that accept key-value pairs as input and generate key-value pairs as output.
Transforms can insert key-value pairs on the local PE or on a remote PE.
Transforms are lazily evaluated (mostly, the exception being transforms that push data to remote PEs).
Keys are versioned and single-assignment.

```
herds_key_t output = {0, 0, 3};
herds_key_t inputs[2] = {{0, 0, 0}, {0, 0, 1}};

ctx->transform(output,
           herds_key_config_t(N * sizeof(int64_t)),
           [N] (...) {
             int64_t *a = (int64_t*)inv[0].get_ptr();
             int64_t *b = (int64_t*)inv[1].get_ptr();
             int64_t *c = (int64_t*)outv[0].get_ptr();
             for (int i = 0; i < N; i++)
                 c[i] = a[i] + b[i];
           }, inputs, 2);
```

Simple vector addition example on CPU

```
herds_key_t output = {0, 0, 3};
herds_key_t inputs[2] = {{0, 0, 0}, {0, 0, 1}};

ctx->transform(output,
           herds_key_config_t(N * sizeof(int64_t)),
           N,
           [N] __device__ (...) {
             int64_t *a = (int64_t*)inv[0].get_ptr();
             int64_t *b = (int64_t*)inv[1].get_ptr();
             int64_t *c = (int64_t*)outv[0].get_ptr();
             c[i] = a[i] + b[i];
           }, inputs, 2);
```

Simple vector addition example on GPU

Distributed HERDS

HERDS can be run shared memory or distributed memory.

During distributed execution:

- Inter-process coordination happens over Conveyors and OpenSHMEM
- Key values are unique within a process, but not across processes (i.e. key `(0, 0, 1)` can exist on rank 0 and rank 1 while referencing logically different objects)
- All movement of keys and values between ranks done explicitly through the APIs below.

API	Description
`comm_region`	Creates a program region, where all communication and computation created within the region will be completed before exiting the region.
`transform`	Insert a transform in a remote PEs execution graph (similar to a lazily executed active message)
`transfer_to_remote`	Transfer a local key-value pair to a remote PE (lazily evaluated on the remote).
`transfer_from_remote`	Transfer a local key-value pair from a remote PE (lazily evaluated locally)
`concat`	Collect the values for N remote keys together and store them as a single value locally.

Example Application (randperm)

bale randperm benchmark (https://github.com/jdevinney/bale)

- High level: Given an array of length N containing the numbers [0, N), produce a randomly permuted array that contains the same values but shuffled in to a random order.

- One possible distributed implementation (throwing darts):
  - Divide input array in to as many chunks as there are PEs
  - For each element in the local chunk of a PE, pick a random PE and “throw a dart at it” (i.e. send that element to the random PE)
  - Each PE collects the darts/values thrown at it in to a randomly shuffled array.
  - Output is the concatenation of all PE’s shuffled arrays.

Example Application (randperm)

```
ctx->comm_region([pe, npes, darts_per_pe, l_N, ctx] {
  for (int64_t i = 0; i < l_N; i++) {
    // Pick a random target PE
    int target_pe = rand() % npes;

    // Send the ith local value for this PE to that
    // PE and store it with key DART_KEY(…)
    ctx->transform(target_pe,
       DART_KEY(pe, darts_per_pe[target_pe]),
       herds_key_config_t(sizeof(int64_t)),
      [pe, l_N, i] (herds_key_t *ink,
          herds_val_t* inv, size_t nin,
          herds_val_t* outv,
          herds_nested_ctx& ctx) {
        int64_t* out =
          (int64_t*)outv[0].get_ptr();
        *out = pe * l_N + i;
      }, NULL, 0);

    // Increment a local counter of how many darts
    // we've sent each PE
    darts_per_pe[target_pe] += 1;
  }

  for (int p = 0; p < npes; p++) {
    // How many darts have we sent PE p?
    int64_t darts = darts_per_pe[p];

    // Tell PE p how many darts we've sent it
    ctx->transform(p, DARTS_FROM_PE_KEY(pe),
      herds_key_config_t(sizeof(darts)),
      [darts] (herds_key_t* ink,
          herds_val_t* inv, size_t nin,
          herds_val_t* outv,
          herds_nested_ctx& ctx) {
        int64_t* out =
          (int64_t*)outv[0].get_ptr();
        *out = darts;
      }, NULL, 0);
  }
});
```

Example Application (randperm)

```
size_t total_local_darts = 0;
std::vector<herds_key_t> concat_keys;

for (int p = 0; p < npes; p++) {
    int64_t nreceived;
    // Copy the number of darts this PE received from PE p in to nreceived using fetch()
    ctx->fetch(DARTS_FROM_PE_KEY(p), &nreceived, sizeof(nreceived));

    // Accumulate the keys for all received darts in to a list
    for (int64_t i = 0; i < nreceived; i++) {
        concat_keys.push_back(DART_KEY(p, i));
    }

    total_local_darts += nreceived;
}

// Use concat to concatenate all received darts in to one value/array
ctx->concat({2, 0, 0}, concat_keys);
```

Example Application (randperm)

```
herds_key_t inputs[] = {{2, 0, 0}};

// Randomly shuffle our locally received darts
ctx->transform({3, 0, 0},
    herds_key_config_t(total_local_darts * sizeof(int64_t)),
    [total_local_darts] (herds_key_t* ink, herds_val_t* inv, size_t nin,
            herds_val_t* outv, herds_nested_ctx& ctx) {
        int64_t* in = (int64_t*)inv[0].get_ptr();
        int64_t* out = (int64_t*)outv[0].get_ptr();

        for (int i = 0; i < total_local_darts; i++) {
            int j = i + rand() % (total_local_darts - i);
            // swap i and j
            out[i] = in[j];
            out[j] = in[i];
        }
    }, inputs, 1);

// Transfer final, shuffled array back to host address space using fetch()
int64_t* lperm = (int64_t*)malloc(total_local_darts * sizeof(*lperm));
assert(lperm);
ctx->fetch({3, 0, 0}, lperm, total_local_darts * sizeof(*lperm));
```

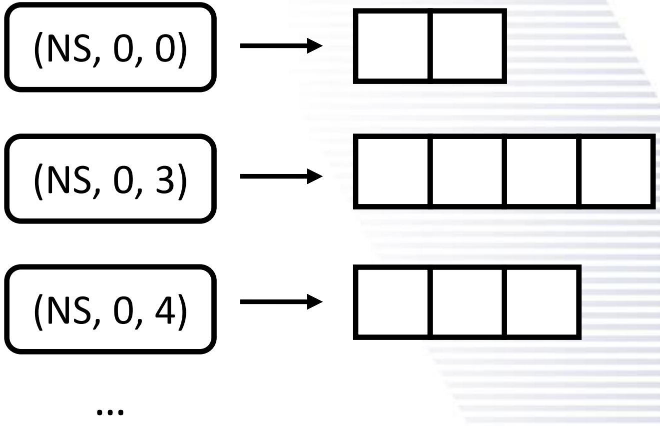
Sparse Matrix Extensions

herds_spmat offers a sparse matrix abstraction over HERDS, parameterized by:

- # rows/columns
- Namespace: All keys created to store data for the sparse matrix are created under the {NS, *, *} namespace

Each row stored as a separate key-value pair.

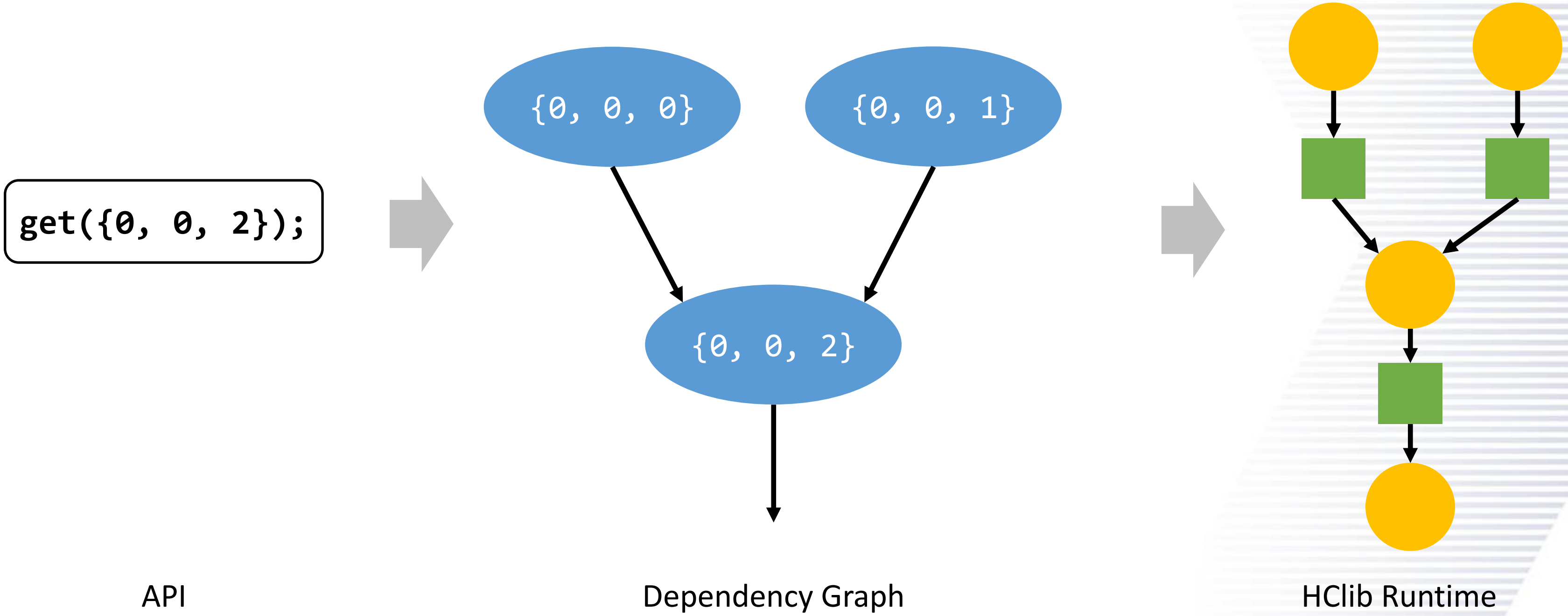
Hides complexity of HERDS key-value APIs with optimized implementations of common sparse matrix operations.



```
herds_spmat* mat = herds_spmat::gen_erdos_renyi_graph_triangle_dist(n, ns,
        p, unit_diag, lower, seed, ctx);
mat = mat->transpose();
mat->update_row(row, cols, nnz);
val = mat->get(r, c);
```

HERDS Runtime

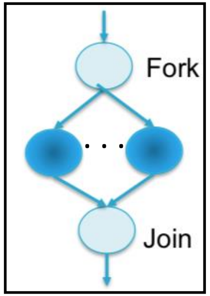
Sparse/dense hash maps are used to store mappings from keys to the logic needed to compute them. HClib is used to schedule and coordinate all computation/communication needed to compute values.



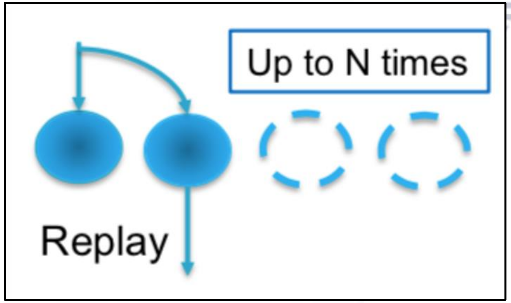
```
get({0, 0, 2});
```

HERDS Runtime

- Heterogeneity
  - Supports single CUDA device per process
  - Abstractions are intended to be flexible enough to support other chips (some exploration of Mellanox Bluefields, but never integrated to runtime)
- Resilience
  - Implemented both replication-based and replay-based resilient tasks
  - Replication: At kv-pair creation, programmer specifies the number of times this value should be replicated
    - Runtime automatically schedules duplicate tasks and validates binary equivalence of outputs
  - Replay: At kv-pair creation, programmer specifies logic for validating value
  - Runtime automatically schedules validation
  - In case of failure of any validation, runtime automatically retries work + re-validation



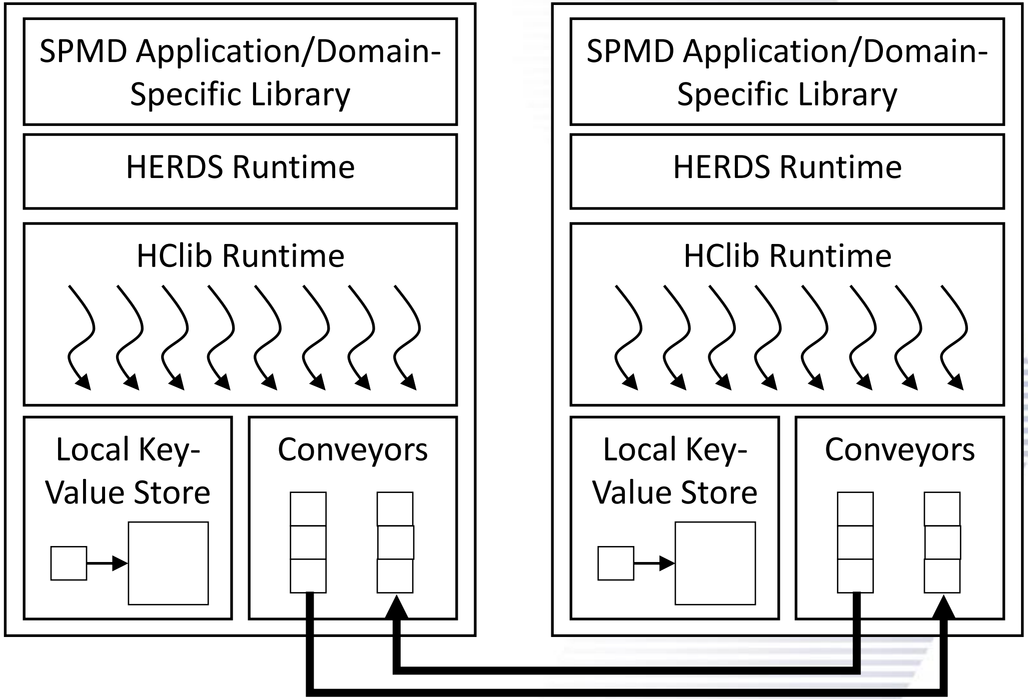
Task replication
(N-way)



Task replay
(up to N times)

HERDS Runtime

- Distribution
  - Leveraging the Conveyors library for efficient inter-node communication over OpenSHMEM
    - Elastic conveyors with progress option.
  - Lowest level APIs offer SPMD model where transformations can be sent to remote PEs (active message).
  - HERDS pushes/advances conveyor when a new remote transformation is launched.
  - HERDS pulls/advances conveyor inside a low priority, yielding task that is scheduled periodically.

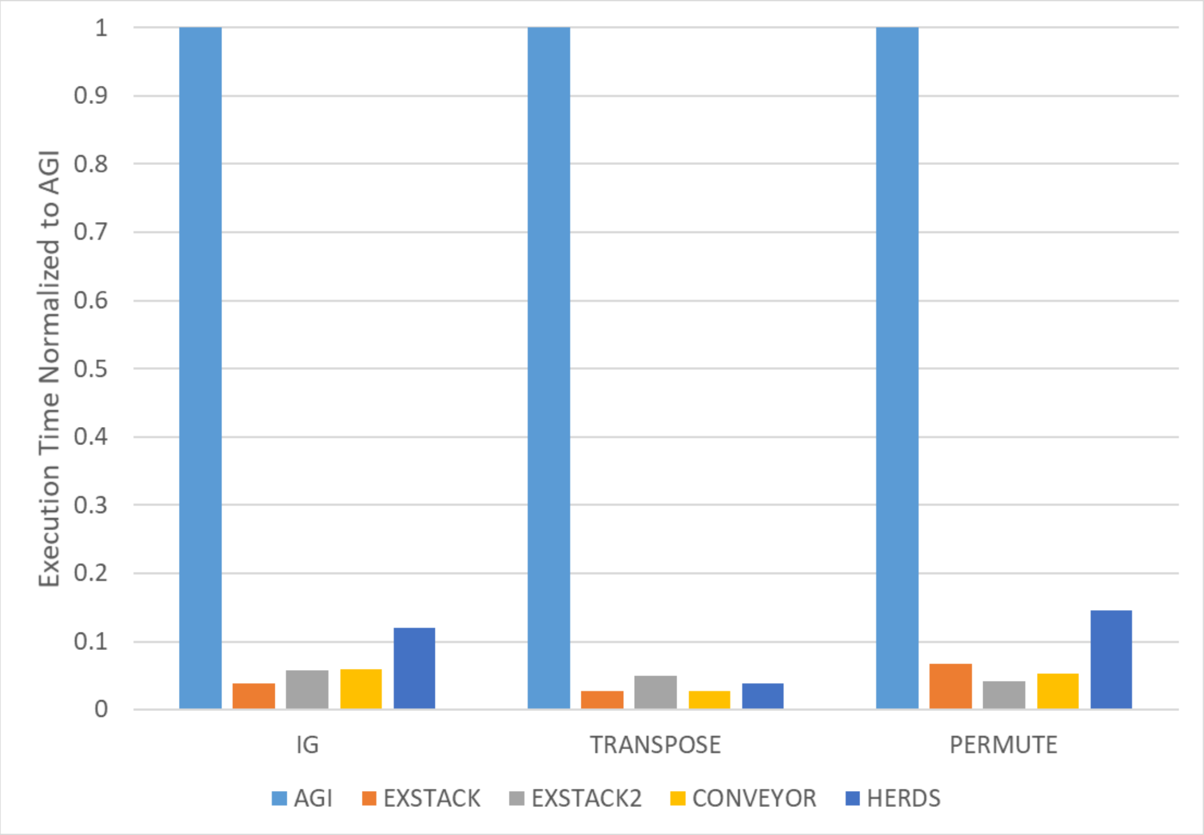


Performance Evaluation

672 PEs (16 nodes on Summit, 42 PEs per node).
Cray OpenSHMEMx.

AGI = Implementation that communicates at the natural granularity of the problem.

Exstack, Exstack2, Conveyors = Aggregating communication runtimes, higher throughput on modern networks



Wrap Up

HERDS uses key-value pairs as its core data abstraction – enables resilience, replication, and a flexible programming model.

HERDS defines values as the output of a transformation applied to some number of other input key-value pairs.

Layers that on top of high performance runtimes for asynchronous tasking and asynchronous communication.

Hides programming model complexity under domain-specific libraries (if desired).

https://github.com/agrippa/herds

Let Max know if you'd like to be added (max.grossman@gatech.edu).