HCL: Distributing Parallel Data Structures in Extreme Scales

Hariharan Devarajan, Anthony Kougkas, Keith Bateman, Xian-He Sun
Illinois Institute of Technology, Department of Computer Science
hdevarajan@hawk.iit.edu, akougkas@iit.edu, kbateman@hawk.iit.edu, sun@iit.edu

***Abstract*—Most parallel programs use irregular control flow and data structures, which are perfect for one-sided communication paradigms such as MPI or PGAS programming languages. However, these environments lack the presence of efficient function-based application libraries that can utilize popular communication fabrics such as TCP, Infinity Band (IB), and RDMA over Converged Ethernet (RoCE). Additionally, there is a lack of high-performance data structure interfaces. We present *Hermes Container Library* (HCL), a high-performance distributed data structures library that offers high-level abstractions including hash-maps, sets, and queues. HCL uses a RPC over RDMA technology that implements a novel procedural programming paradigm. In this paper, we argue a RPC over RDMA technology can serve as a high-performance, flexible, and co-ordination free backend for implementing complex data structures. Evaluation results from testing real workloads shows that HCL programs are 2x to 12x faster compared to BCL, a state-of-the-art distributed data structure library.**

***Index Terms*—Distributed Data Structure, RPC over RDMA, Hybrid Data Access Model, HPC Data Containers**

I. Introduction

Applications that include complex data distribution and irregular control flows are extremely complex to write [1], [2]. To build parallel programs, scientists employ various parallel programming models such as Partitioned Global Address Space (PGAS) [3], Message Passing Interface (MPI) [4], or task-based paradigms [5], [6]. However, all of these provide application developers with primitive constructs on top of the hardware that are quite complex to program with and highly error-prone. Many parallel languages [7], [8], [6] and libraries [9], [10], [11], [12], [13] have been introduced to ease this complexity. These parallel languages provide abstractions with low-level control flows such as parallel access, task creation, messaging layers, etc. However, to fully leverage the power of such parallel languages, developers need to write the entire program natively in one such language. Thus, legacy code and software stacks need to be rewritten in order to utilize parallel languages. In contrast, parallel programming libraries extend current serial compilers with parallel programming abstracts. This leads to a less intrusive approach where application developers can still write programs in serial languages which are extended with additional functionality. Scientific productivity can be increased by parallel programs that utilize high-level clean abstractions provided by these libraries.

One such parallel programming library, the Berkeley Container Library (BCL) [11], that was recently introduced, is intended to support applications with irregular computation and communication patterns. Specifically, BCL provides high-level portable data structures, such as hashmaps and queues, with asynchronous access to structures distributed across multiple processes. BCL aims to steer away from a bulk-synchronous programming (BSP) model [14] by providing a complementary set of C++ STL-like abstractions. It uses one-sided communication primitives that can be executed by RDMA-capable hardware [15], and, thus, eliminates the requirement of remote CPU coordination. Consequently, instead of a low-level remote read and write operation consistent with the PGAS model, BCL offers higher level primitives such as insert and find (in a hashtable). To achieve this, BCL employs three core architectural principles: a) use of client-side memory management and Compare-And-Swap (CAS) operations [16] to ensure consistency, b) all data structures are partitioned to ensure good locality, and c) the underlying communication layer should be abstracted, as long as it supports one-sided Remote Memory Access (RMA) [17]. BCL demonstrates how high-performance implementations of these data structures make the development of parallel applications straightforward while matching the performance of specialized parallel languages.

Distributing data structures involves partitioning a data container and placing it on a collection of servers. To ensure data consistency, synchronization is needed. In an effort to reduce the need for synchronizations between the caller and target processes, BCL implements such a data distribution by leveraging one-sided RMA operations. Hence, BCL uses a client-side programming model where all data interactions on the remote data container are solely instructed by the clients. However, this architectural choice implies several performance and programmability limitations as follows: a) increased network congestion caused by multiple remote calls made by clients per operation. For example, for each `insert()` operation, the client needs to perform two remote CAS operations before it can insert the data. b) Low write asynchronicity caused by the necessity of performing a flush operation, which forces the callers to serialize updates. c) Limited operation concurrency caused by memory region locking by the CAS instructions. d) Increased operation latency caused by multiple remote calls which a client makes in order to locate the next available slot in the data container. For example, for all hash-based structures, bucket collision is resolved by the client retrying until it finds the next available bucket. e) Increased cost of re-balancing a data structure, which is caused by a static pre-

978-1-7281-6677-3/20/$31.00 ©2020 IEEE
DOI 10.1109/CLUSTER49012.2020.00035

allocated partitioning that the clients must agree upon. This causes either increased client all-to-all synchronizations or an over-provisioning of the available resources to minimize frequent re-hashing. f) Under-utilization of system resources, which is caused by imposing a static predefined data entry size.

In this paper, we argue that a combination of a Remote Procedure Call (RPC) protocol and Remote Direct Memory Access model (e.g., RDMA or RoCE) can alleviate the above challenges and offer an infrastructure to design high-performance, scalable, and highly concurrent distributed data structures (DDS). The RPC protocol implies a level of location transparency (i.e., remote functions appear as local) enabling network programming that follows a request–response model. In other words, the RPC protocol is a form of inter-process communication (IPC) and can support server-side callback functions. However, RPC protocols demand the participation of the remote CPU (i.e., interrupting the destination core) resulting in a slower throughput. The key insight this paper offers is that RDMA technology can alleviate this issue by offloading the RPC instructions to the NIC, conserving valuable CPU resources. Note that the computation capabilities of modern NICs (e.g., Mellanox™ BlueField2) are significantly improved due to multi-core designs.

In this work, we present Hermes Container Library (HCL): a new, high-performance, and scalable distributed data structure library that uses an RPC-over-RDMA approach as its functional communication fabric. HCL aims to provide high-level data structures in the form of STL containers, such as ordered/unordered maps and sets, as well as priority and FIFO queues, while abstracting the low-level details of distribution, communication, and data access semantics and providing coordination-free data access by offloading the instruction execution to the RDMA hardware using the RPC-over-RDMA approach. HCL can support highly parallel workloads with irregular patterns, indexing services, scheduling, data sharing, and process-to-process lock-free synchronizations. Consistent with any PGAS implementation, HCL data structures reside in a global address space where multiple processes can access data concurrently. HCL uses an abstract serialization template, called DataBox, that defines a standard methodology to transparently serialize complex data-types. HCL uses the Open Fabric Interface (OFI) to build a portable cross-platform communication fabric able to interface with any underlying network protocols (e.g., IB, TCP, CC, etc.). Additionally, HCL distinguishes between node-local and inter-node client accesses, resulting in a hybrid access model, in order to further optimize the overall throughput of its DDSs. Lastly, acknowledging the need to persist ever-growing critical data, HCL DDSs can reside in storage-class memory technologies via the use of a memory-mapped backing file. The contributions of this work are:

1) Design and implementation of RPC-over-RDMA protocol, that enables high performance and co-ordination free communication fabric for procedural programming paradigm.
2) A distributed data structure library built for high performance using RPC-over-RDMA protocol.
3) A hybrid data access model, efficiently and simply making decisions about whether or not to perform RPC calls based on data locality.

II. Background and Motivation

A. Distributing a Data Structure

Many approaches have been proposed to distribute primitive data structures. PGAS languages such as Chapel [7], Fortress [18], X10, Titanium [8], HPF [19], and ZPL [20] offer distributed memory data structures such as multidimensional, dense or sparse arrays. However, rich data structures, which are present in sequential environments, are not supported due to a more complex concurrent access model that partitioned arrays simply cannot express. Tiling is a popular technique to provide concurrency control, where a group of processes operate on a tile, but lose the ability to enforce data locality optimizations that most irregular applications would need to run faster. The lack of support for data structures at a higher level of abstraction (e.g., hash-tables or queues) hurts the programmability and the performance. Developers are forced either to work with low-level control structures or to implement a custom solution using basic network primitives such as MPI send-receive or even one-sided RMA. This process is complex, error-prone, non-portable, and not scalable.

B. Client-side Programming

One proposed solution to provide high-level, high-performance, and robust data structures to parallel programs is the BCL [11] cross-platform distributed data structures library. In BCL, the clients expose a memory segment into the global shared memory window and agree on its management via global pointers. Remote reads and writes are then executed, in a coordination-free way, by using the RDMA hardware instead of the destination CPUs. At its core, BCL requires the support of remote memory operations and atomics (compare-and-swap (CAS)) from the network hardware and software stack. Without CAS support, BCL structures cannot be implemented. This is because BCL follows a client-side programming paradigm where the client needs to execute all necessary instructions to interact with a remote data structure. For example, take a client that wants to insert an element to a BCL hashmap. First, the client needs to check the bucket state and reserve it via a CAS operation. If this reservation fails, the client will retry on the next bucket in sequence. Once the reservation succeeds, the client will write the data in the bucket and set the state of the bucket to ”ready” for future accesses via a CAS operation. This whole process can be very quick but it can also be very inefficient under high-concurrency scenarios where multiple clients are trying to push data to the hashmap, and, thus, CAS operations have to be serialized behind a lock of the same memory region.

C. A Motivating Example

The client-side programming approach of DDS manipulation, as proposed in BCL, can be limiting in performance in parallel and highly-concurrent environments due to multiple remote CAS operations [21]. We argue that BCL can be

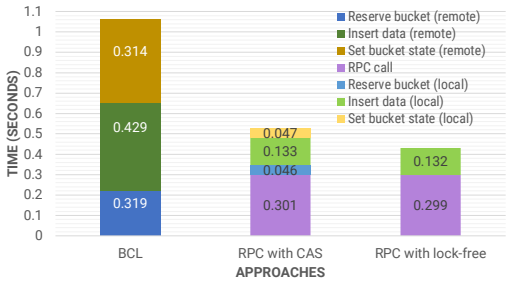


Fig. 1. Motivating test case.

enhanced and extended with additional functionality if we move to a procedural programming paradigm where all necessary instructions (and data) to manipulate the remote memory region can be bundled inside an RPC call and executed on the destination node. By doing so, we can reduce the number of network calls (i.e., multiple client-to-server roundtrips) and eliminate any locking imposed by remote CAS operations, and, thus, a performance gain will be observed. To test our hypothesis and quantify the cost of BCL's client-side management of remote memory regions, which uses multiple remote CAS operations to modify the data structure, we conduct the following test. We use 40 clients on one node to issue 8192 insert() calls of 4KB each. The target hashmap partition resides in a different node requiring remote network calls. For BCL the following three operations for each client are needed to complete an insert: a) CAS to reserve the hashmap bucket, b) RDMA write to put the data in the bucket, and c) CAS to set the new bucket state to ready. For the procedural programming approach, we bundle all three operations to an RPC call and invoke the execution on the remote RDMA NIC. As an extra optimization, we implemented a lock-free implementation of hashmap that eliminates the need of CAS operations. This can only be done under the procedural programming approach since it requires a dynamic allocation and destruction of remote address space [22]. We measure the average time in seconds that each client process needs to complete all 8K inserts. Figure 1 shows the results of this test. As can be seen, the total average cost for BCL's client-side approach is 1.062 seconds. If broken down to the three operations, we see that the remote CAS operations impose a significant burden to the client consuming about 2/3 of the overall time. In contrast, the procedural programming approach is 2x faster since, once the RPC call has been invoked, all CAS operations are executed locally (i.e., local memory performance). The cost of the remote RPC call is about 0.30 seconds and is consistent with the network performance between the two test nodes. Lastly, the lock-free implementation removes the need for CAS operations, and, thus, is faster than BCL by 2.5x. This simple hypothesis motivates us to propose a change in paradigm from client-side imperative to procedural programming with RDMA in order to further optimize distributed data structure access, concurrency control, and locality.

III. Hermes Container Library

The Hermes Container Library (HCL) is a cross-platform data structure library (https://bitbucket.org/scs-io/hcl/) that implements high-level distributed data structures. It follows the PGAS memory model but is based on procedural programming, which uses remote function invocation to manipulate remote memory. During initialization, one or more processes in the node can create a shared memory segment that other processes (both local and remote) can read and write to by invoking functions. Each function contains a caller identifier (i.e., where the function is invoked) and an operation to execute at the target memory segment. Together, these two values can uniquely identify operations that can be performed on the global address space. Each function can manipulate the remote memory address based on the operation it performs. These functions can be invoked synchronously, asynchronously, or through callbacks on other functions. This enables a rich set of functionality support required by any data structure. The users can include the HCL library header and utilize the data structures by calling the constructor.

A. HCL Design Challenges

1) How can remote function invocation be supported: HCL handles all remote memory region accesses (and mutations) through invocation as a core principle of data manipulation. Each invocation includes two identifiers: a) the target process (i.e., destination) and b) the operation to be executed. Invocations are implemented by an RPC call which bundles the instructions (and data if needed) and ships them to the target process for execution. Upon completion of the invocation, the target process returns the response.

2) How can atomic operations on DDS be supported: HCL inherently offers atomic operations [23] by supporting remote function invocation mechanisms and compare-and-swap operations. HCL depends on the OS kernel to provide high-quality interfaces to atomic operations as implemented in the hardware. When such hardware support is not present, HCL also supports atomics via mutexes.

3) How can data consistency be supported: Where required/instructed, HCL can provide data consistency between concurrent access to the share memory region regardless of its origin (i.e., CPU or NIC core). It achieves this by utilizing a lock-free and consistent local data structures, much like [24], which are the building block of DDSs within HCL.

4) How can data durability for DDS be supported: All HCL data structures can be protected against faults by either backing them up to a durable medium such as an NVMe drive or by replication. HCL can map the memory segments to a memory mapped file and let the kernel synchronize (i.e., flush) the contents of the mapped memory region to the file. Additionally, HCL can enable replication on its data structures. Each operation is hashed to a server where the invocation is executed. Replication occurs asynchronously at the server side, where the target process will further hash an operation to more servers.

B. RPC over RDMA

HCL uses a new communication framework that implements the traditional RPC protocol over an RDMA-enable network. This framework avoids the pitfalls of a traditional CPU-based RPC protocol [25] by using the RDMA infrastructure to

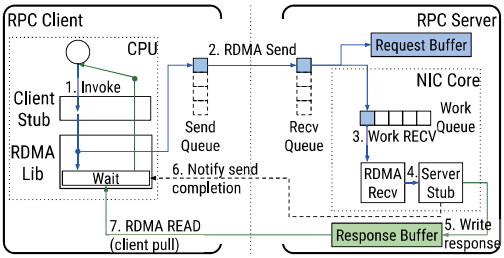


Fig. 2. RPC over RDMA design.

limit interruptions and other overheads, and thus, offer a high performance communication layer for data structure access. HCL's RPC-over-RDMA (RoR) framework (shown in Figure 2) modifies the client and server stubs to enable the RDMA functionality. Initially, users submit their functions by calling the `bind()` method that maps them to an RPC invocation registry. When clients call `invoke()`, the client stub inserts the request to a *request buffer* (residing at the server's main memory) using an RDMA_SEND. Once the request arrives in the request buffer, the RPC server, which is running on the NIC core, pulls the request using IBV_WC_RECV available by the RDMA work-queue [26] mechanism. The request is then processed using the server stub that de-marshals it and executes the invoked function. The response is placed in a *response buffer*. Finally, the client stub using the ibv_get_cq_event gets notified about completion of send and it pulls the results using IBV_WR_RDMA_READ from the response buffers. HCL's RoR framework enables two key innovations: *request aggregations* and a *client-pulling* response paradigm. First, to minimize remote calls and network traffic, the RoR framework processes incoming requests on the server's NIC which exposes the opportunity to aggregate multiple instructions before execution. Second, the framework employs a data pull model where results are remotely fetched by the clients through an RDMA read operation instead of being sent by the server as described above. HCL's RoR framework enables fast execution of RPC calls using the NIC core (i.e, without the involvement of target CPUs).

C. The HCL DataBox Abstraction

To transparently manipulate a DDS in the global address space, HCL introduces the DataBox abstraction. A DataBox is a template that provides mechanisms for defining, serializing, transmitting, and storing complex data structures. A DataBox offers a higher-level encapsulation of complex data types (e.g., class or struct), which are not byte-copyable. For instance, a C++ standard class might include pointers to external objects that do not carry a meaningful interpretation outside the scope of the source process (i.e., source memory becomes inaccessible). This is also true for supporting newer byte-addressable mediums such as NVRAM or PCM. DataBoxes do not use serialization for simple byte-copyable data types.

1) DataBox Transmission (RPC over RDMA): DataBoxes contain a transmission mechanism defined by an RPC over RDMA protocol. In this case, the server process runs on one of the RDMA NIC's cores and the transfer of arguments happens using a one-sided access [27]. The benefit of this design is that the CPU core is no longer participating in the function execution. However, due to the limited computational capability of the NIC's cores, the invoked functions have to be lightweight. This fits perfectly with the case of a data structure library where most operations are not computationally intensive (i.e, the structure can be manipulated with a small number of instructions). HCL implements this by spawning the server-stub on the RDMA-NIC through an asynchronous work-queue mechanism. The serialized DataBox is submitted, using the OFI's ibverbs, to the target work-queue where a NIC core will process it. HCL utilizes the RDMA work-queue technology to provide a multi-threaded asynchronous listening and handling of all transmitted DataBoxes. Additionally, the RPC protocol inherently allows HCL to support callback functions. Hence, HCL's DataBoxes combine the benefits of both UPC++, which uses high-level procedural programming, and BCL, which uses one-sided communications.

2) DataBox Serialization: The DataBox abstraction are implemented using a C++ generic system. A DataBox is a class template that provides a specification for implementing complex data types. To transmit or store its data, a DataBox provides a standard serialize/deserialize method which can use different serialization libraries in the backend, since different serialization libraries excel in different environments. For instance, most supercomputers might provide popular serialization as a module. HCL currently supports three high-performance libraries [28] as a backend: MSGPACK, Cereal, and FlatBuffers. DataBoxes can serialize a fixed or variable length object. This distinction is handled during the compile-time of the application. Also, for their own data types, users can define their own custom serialization function which is resolved dynamically during runtime. Lastly, HCL provides native support for standard STL containers.

3) DataBox Event Handling via Callbacks: Callback methods can be implemented by the procedural programming paradigm HCL employs. This can be achieved using a identifier to a remote function which is executed after the main data structure operations. These are extremely powerful in cases where we want to aggregate multiple data-local operations together. This is enabled by mapping several spatially located updates to be performed with one call by chaining multiple callbacks on a single function invocation. HCL callbacks can be a powerful and expressive tool, and are readily available to developers. Using callbacks can enable a conditional execution of multiple operations in one call which minimizes the network congestion, and, thus, optimizes performance.

4) DataBox Asynchronous RPC: All HCL data structures support asynchronous operations by default. Each function invocation creates a future object (much like C++ future and wait operations), which gets the response after the call is executed. Thus, providing synchronous and asynchronous models is a matter of timing when the caller waits for the future object. In synchronous execution, the caller blocks waiting for the response immediately after making the invocation call, whereas, in asynchronous execution, the caller can come back at a later point and check the future

object's status for the response. Asynchronicity increases overlaps with other computations and the use of concurrent communication lanes within the hardware, thereby enabling efficient collectives (e.g., broadcast, all gather/scatter).

5) DataBox Hybrid Data Access Model: Since all DDSs are partitioned across several nodes, DataBox's data access model distinguishes between *intra-node* and *inter-node* accesses (i.e., local or remote). Modern CPU core count has grown significantly where each node runs hundreds of processes. HCL optimizes intra-node accesses by directly exposing data from the local partition without the need of an RPC call. Invoking RPC calls for local partitions incurs unnecessary costs including the processing of the RPC call and the network congestion on the local NIC. By avoiding this cost, HCL can significantly boost performance and system utilization across all operations. However, remote partitions are normally exposed through the DataBox transmission mechanism. Thus, HCL's DataBoxes enable a hybrid data access model that is achieved by setting a locality flag during initialization. In other words, if the target process has the same nodeID as the caller-process, then a Direct Memory Access (DMA) call is made.

6) DataBox Persistency: HCL can extend the PGAS model to include persistent devices (e.g., NVRAM, NVMe-SSD, HDD), and thus, offer a unified memory and storage address space. To achieve this, DataBoxes implement a shared memory model [29] on top of these devices, where regions of these devices are memory-mapped. These memory-mapped regions are synchronized with the underlying device on a per-operation basis, which ensures all data is always present in the device. However, this synchronization can be configured to be relaxed, which would mean performing these synchronizations in the background. This tunes the performance of the DDS based on the application's requirements. Therefore, DataBoxes can offer persistence for all DDSs by mapping a data structure partition to both a node and a medium. This is an essential feature as non-volatile memory technologies keep growing; it is imperative to extend the data structure library to utilize these new mediums in order to achieve DDS durability.

D. HCL Data Structures

HCL offers a plethora of C++ STL-like distributed data containers grouped into three major categories: maps, sets, and queues. For maps and sets, HCL offers both unordered (e.g., dictionary in Python or hashmap in Java) and ordered versions. For queues, HCL offers both standard FIFO and priority versions. HCL's compatibility to the Standard C++ library boosts productivity since developers are already familiar with the standard containers. HCL takes these abstractions and distributes them across a collection of nodes in a cluster. All distributed data structures (DDS) in HCL are globally visible (i.e., public) since they reside in globally addressable memory shared across all participating nodes. HCL's data distribution model is based upon a partitioning technique where a data structure can be hosted by one or more memory regions (which are still globally visible) based on its ordering properties. For instance, an unordered_map can be placed on all nodes

```
auto sort(const std::vector<int>& data) {
  std::vector<std::vector<int>> buffers(BCL::nprocs());
  std::vector<BCL::FastQueue<int>> queues;
  for (size_t rank = 0; rank < BCL::nprocs(); rank++)
      queues.push_back(BCL::FastQueue<int>(rank, queue_size));
  for (auto& val : data) {
      size_t rank = map_to_rank(val);
      buffers[rank].push_back(val);
      if (buffers[rank].size() >= message_size) {
          queues[rank].push(buffers[rank]);
          buffers[rank].clear();
      }
  }
  for (size_t i = 0; i < buffers.size(); i++)
      queues[i].push(buffers[i]);
  BCL::barrier();
  std::sort(queues[BCL::rank()].begin().local(),
  queues[BCL::rank()].end().local());
  return queues[BCL::rank()].as_vector();
}
```

```
std::vector sort(const std::vector& data) {
  auto set = hcl::set<int,std::string>();
  for (auto& val : data) {
      set.insert(val);
  }
  return set.local()
}
```

Fig. 3. Example of Sorting using BCL and HCL.

since elements are not bound together in any particular order. In contrast, a distributed queue cannot be split into multiple partitions since it will violate the ordering property between its elements. In summary, HCL categorizes its containers into single- and multi-partitioned data structures. As a special case, ordered structures are built using multiple single-partitioned structures that are abstracted behind a global interface.

Since all data structures implement HCL's DataBox abstraction, they have several commonalities. First, all DDSs support complex data types and their entries can be of variable-length. For user-defined types DataBox provides template definitions for custom data serializations. Most common types, however, are handled automatically without any intervention by the user. Second, all DDS operations are lock-free and do not require any global synchronization, even in the case of collective operations such as initializing or resizing a DDS, which reduces the associated costs. This approach allows HCL to support a Multiple-Writer-Multiple-Reader (MWMR) access model which increases performance via concurrency. This allows HCL to have heterogeneous partitions within PGAS, and to enable dynamic addition/removal of partitions. Third, all HCL DDSs support user-defined comparators and operators (using std::hash and std::less) for defining custom data distributions and/or ordering. Lastly, all DDS operations are inherently atomic due to the HCL's functional paradigm. Further, HCL allows its users to tune the level of atomicity by setting the appropriate concurrency control parameter. Table I provides an overview of all HCL's data structures, their operations, and their complexity. As demonstrated by the table, each high-level data structure operation is compiled down to only one remote invocation and a few local operations.

1) Hash Map and Hash Set: All hash data structures (i.e., both maps and sets) are implemented as a single logically contiguous array of buckets distributed block-wise among multiple partitions in the global address space. Each bucket is a struct consisting of a key and a value for maps and a key for sets. HCL hash structures use two levels of hashing: one for choosing the block of the global partition, and one to locate a bucket within the specified partition. By default, HCL uses the C++ `std::hash<K>` template struct in the standard namespace. Users can override this template in case they require a different distribution of keys. We employ a lock-free Cuckoo Hash algorithm [30], which allows multiple insertions on the same key to be always consistent, resolves cache collisions using a secondary array of buckets, and utilizes concurrency to increase write performance.

Data Structure	Operations	Description	Cost (Time Complexity)
HCL::unordered_map	bool insert(const K &key, const V &val)	Insert item into hash table	F+L+W
	bool find(const K &key, V &val)	Find item in table, return val.	F+L+R
	bool resize(const int partition_id, const int new size)	Resize the hash table at the partition	F+N(R+W)
HCL::map	bool insert(const K &key, const V &val)	Insert item into ordered map	F+L(log(N)) +W
	bool find(const K &key, V &val)	Find item in map, return val.	F+L(log(N)) +R
	bool resize(const int partition_id, const int new size)	Resize the map at the partition	F+Nlog(N)(R+W)
HCL::unordered_set	bool insert(const K &key)	Insert item into hash set	F+L+W
	bool find(const K &key)	Find item in set, return if exists.	F+L+R
	bool resize(const int partition_id, const int new size)	Resize the hash set at the partition	F+N(R+W)
HCL::set	bool insert(const K &key)	Insert item into ordered set	F+L(log(N)) +W
	bool find(const K &key)	Find item in set, return if exists.	F+L(log(N)) +R
	bool resize(const int partition_id, const int new size)	Resize the set at the partition	F+Nlog(N)(R+W)
HCL::queue	bool push(const T &val)	Push element into queue	F+L+W
	bool pop(const T &val)	Pop element from queue	F+L+R
	bool push(const std::vector<T>&vals)	Push multiple elements into queue	F+L+E*W
	bool pop(const std::vector &vals, const size_t &E)	Pop multiple elements from queue	F+L+E*R
HCL::priority_queue	bool push(const T &val)	Push element into queue	F+Llog(N)+W
	bool pop(const T &val)	Pop element from queue	F+L+R
	bool push(const std::vector &vals)	Push multiple elements into queue	F+Llog(N)+E*W
	bool pop(const std::vector &vals, const size_t &E)	Pop multiple elements from queue	F+L+E*R

TABLE I

A SELECTION OF METHODS FROM HCL DATA STRUCTURES. COSTS ARE WORST CASE, USING IMPLEMENTATION CHOSEN WITH CONCURRENT ACCESSES. R IS THE COST OF A LOCAL READ, W THE COST OF A LOCAL WRITE, L THE COST OF A LOCAL MEMORY OPERATION, N NUMBER OF ENTRIES, F IS THE COST OF INVOKING A FUNCTION ON REMOTE MEMORY, AND E THE NUMBER OF ELEMENTS INVOLVED.

HCL's HCL::unordered_map and HCL::unordered_set are distributed data structures. Clients can create instances of unordered maps and sets (by calling the constructor Figure 3), and use them as if they were local STL containers. Both structures start with a default size of 128 buckets and can resize themselves if needed. We use a load factor of 0.75 at which the structure doubles its number of buckets. A `realloc` call is made in case the partition needs to be resized. In the case where this resizing fails, the partition is rehashed with a new memory allocation. This operation is localized to the involved partition and can be either triggered by the user explicitly or automatically when the load factor is reached. HCL's unordered maps and sets execute insertions and lookups as follows: First, the inserting (finding) process computes the appropriate partition by performing a hash operation on the key. If a node-local partition is chosen, the RPC infrastructure is bypassed and insertion (find) is performed on the shared memory (i.e., without involving the NIC). For a remote partition, an insert (find) function is invoked on a target process by an RPC call. In case of inserts, the target process locates an appropriate bucket on its local partition and writes the data. Collisions are resolved asynchronously by the secondary bucket mechanism that the Cuckoo algorithm uses. In case of finds, the target process locates the key locally, from either the primary or secondary, and reads the data.

2) Ordered Map and Set: HCL's ordered data structures, both maps and sets, are each implemented as an ordered partition, containing the key space, which is distributed across multiple partitions in the global address space. The key-space length on each partition is configurable via tuning the locality of the keys. To maintain order within each partition, a lock-free red-black tree [31] algorithm is used due to its ability to support high concurrency and asynchronous conflict resolution (via its Node Lock Protocol (NLP) framework). Each node of the tree represents a key-value pair for maps and a key for sets. Since the red-black tree size increases along with the number of nodes it will eventually fill the entire partition. In this case, HCL triggers a `realloc` to resize the partition and accommodate the new nodes of the tree. If resizing fails, all nodes from the old partition are re-inserted into a new memory allocation which only reflects the memory segment that ran out of space. This whole process can either be triggered by the user explicitly or automatically when a threshold is reached.

HCL's `HCL::map` and `HCL::set` are ordered distributed data structures. Clients can create an instance of `HCL::map`/`HCL::set`, by calling the appropriate constructor, without any need for coordination. By default, HCL uses the C++ comparator function `std::less<K>` template struct in the standard namespace. As before, users can override the template to achieve a different ordering of elements in their data structures. HCL's ordered maps and sets execute insertions and lookups in a similar way as the unordered versions. The main difference is the way these ordered structures handle key conflicts. Ordered maps and sets handle collisions by using a linked list that, in the worst case, performs $O(m + logn)$ per lookup (where n is the number of nodes and m is the length of the linked-list). The length of list is kept constant by using a background relocation technique [31]. Also, instead of random key distribution, the ordered versions of maps and sets distribute the key-space in a round-robin fashion based on the key length.

3) Queues: All HCL queues support the MWMR access model for concurrent push and pop operations. HCL queues are implemented as a single-partitioned structure, but are globally visible. The queues are identified by the process ID that hosts the partition. Queue elements can be of variable length. The queue size can dynamically grow by a resizing of the partition allocation, which involves only the process that hosts the queue. Upon resizing, copy and delete semantics are used to migrate existing data from the previous memory location to the new one, and new incoming push operations are stalled. However, pop operations can still be served during migrations.

Both types of queues share the same invocation methodology, but differ as to how they handle the operations on the target process. HCL offers two types of lock-free queues:

(A) FIFO queues (i.e., `HCL::queue`) use a state-of-the-art algorithm that maintains a list of pointers to allow concurrent lock-free operations [32]. During a push() operation, a new node is added to the list at the current tail by a CAS increment on the tail list position. To optimize acquisitions of the front of the queue, HCL uses a background asynchronous *fix-list* operation to consolidate all the elements based on arrival time, and, thus, maintain consistency of the tail. During a pop() operation, the target process grabs the head of the queue and returns to the client.

(B) Priority queues (i.e., `HCL::priority_queue`) use a lock-free implementation based on a multi-dimensional linked list [33]. It uses the CAS over the multi-dimensional linked list to support concurrent push and pop operations. It uses a background purge methodology to clean up logically invalidated nodes in the linked-list. During a push(), a new node is created and placed in the next node list. To optimize concurrency, HCL uses a background predicate lookup methodology that resolves conflicts based on arrival time and priority. During a pop() operation, the minimum node is located from the top of the multi-dimensional linked list and is marked for deletion. A background process is used to delete all the marked nodes and compact the multi-dimensional linked list.

IV. Evaluation

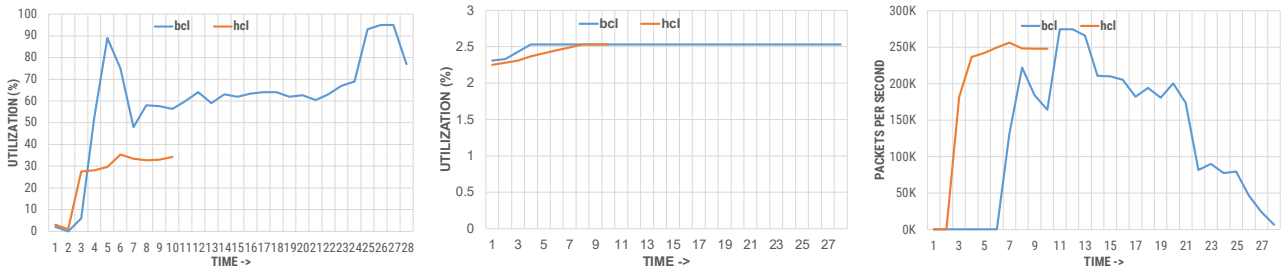
A. Methodology and Experimental Setup

All experiments were conducted on the Ares supercomputer at the Illinois Institute of Technology. Each compute node has a dual Intel(R) Xeon Scalable Silver 4114 @ 2.20GHz (i.e., 40 cores per node), 96 GB RAM, a ConnectX-4 Lx 1x40GbE QSFP+ Ethernet Adapter with RoCE, and a local 512GB NVMe SSD. The total experimental cluster consists of 2560 client MPI ranks (i.e., 64 nodes). We use CentOS 7.1 as the operating system, and the MPI version is Mpich 3.3.2. To evaluate HCL, we first use a set of synthetic benchmarks to measure the performance of its internal components. The synthetic benchmark is a multi-process program which stresses the components by performing reads and writes to remote/local partitions. We also use a set of micro-benchmarks to evaluate the throughput of HCL DDSs. Finally, we use a set of real-world applications: ISx benchmark [34], an integer sorting mini-application, and Meraculous [35], a large-scale genome assembly application kernel. We compare HCL with BCL configured with GASNet-EX as its communication layer, which was shown [11] to be the fastest communication fabric in these tests. We measure the end-to-end execution time. Also, we measure throughput as the number of operations that can be supported per second. For our tests, we executed each test ten times, and we report the average.

B. HCL Architectural Evaluation

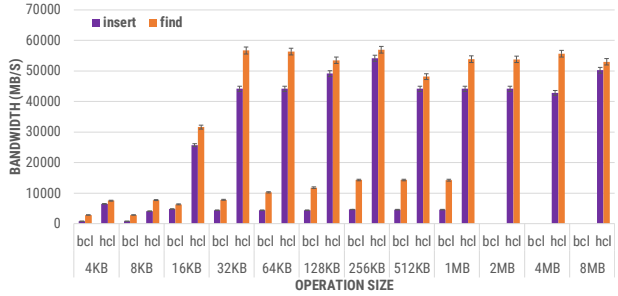
1) RPC-over-RDMA Overhead Analysis: RPC is a very flexible protocol that can provide high-level operations as functional programs. However, traditional RPC incurs overheads [25] on CPU, memory, and network on the node where it runs. An RPC-over-RDMA can theoretically eliminate this overhead by utilizing the RDMA core for its processing. To measure the impact of HCL's RPC-over-RDMA procedural programming approach on the application, we conduct an extensive profiling on system resources (i.e., RDMA NIC core, RAM, and network) using Intel's Performance Analysis Toolkit (PAT) [36]. In this test, we compare HCL's RPC-over-RDMA with BCL's client-side pure RDMA approach by setting up two nodes, one node with 40 client processes and the other with one target partition (i.e., memory segment) exposed. Each client process issues 8192 operations that write 4KB of data to the remote target partition. The results of the test are shown in Figure 4. The x-axis shows the time elapsed in seconds whereas the y-axis shows the CPU utilization, memory utilization, and network performance in each subfigure respectively. BCL completed the test in 28 seconds, and, hence, it has more data points whereas HCL finished in 10.5 seconds.
RDMA NIC core utilization: figure 4(a) shows the RDMA NIC core utilization. For HCL is 33% where as for BCL is around 60% (sometimes spiked at 90%). This is caused by the necessary CAS operations performed by the clients concurrently on the target partition. These CAS operations are served by the RDMA work-queue and are performed atomically. In contrast, the HCL functional approach performs one RPC call on the target partition and all CAS operations are performed locally. This leads to much faster CAS execution resulting in lower RDMA NIC core utilization.
Memory utilization: figure 4(b) shows the memory utilization. For BCL, the memory utilization increases at a constant rate for the first couple of seconds since BCL allocates the memory target partition during the application's initialization. In contrast, HCL manages memory dynamically and initializes the target partition with a smaller size. It expands its size as operations are executed, eventually reaching the same overall memory utilization. This dynamic memory allocation design allows a more efficient memory management and it can boost performance for more concurrent workloads.
Network utilization: figure 4(c) shows the network performance measured by packets sent/received per second. We observe that for the same number of packets, BCL achieves 4x less packet rate (computed using the average packet rate of BCL over HCL) when compared to HCL. This is mostly caused by the client-side CAS operations that are executed remotely compared to HCL that performs these operations on the target partition locally, and, thus, has a better and more stable network performance. As can be seen, BCL is also slower to saturate the network since it spends significant time initializing the remote memory segment (i.e. first 6 seconds).

2) Hybrid Data Access Model: In the PGAS model, each node holds a partition of the global memory address space. As multiple processes are co-located on one node (a rising trend as modern CPUs become denser), accessing the co-located partitions should be optimized. Additionally, inter-node access also has to be optimized to saturate the network capabilities. To

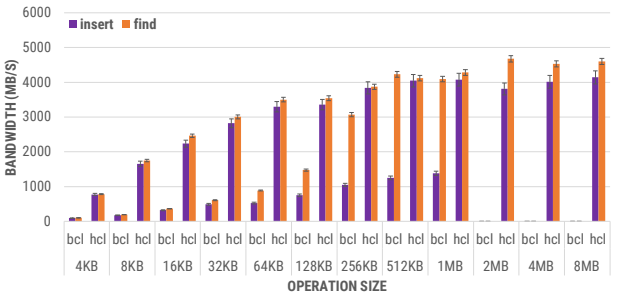


(a) CPU Utilization (b) Memory Utilization (c) Network Performance

Fig. 4. Profiling of HCL and BCL



(a) Intra-node access performance of BCL and HCL



(b) Inter-node access performance of BCL and HCL

Fig. 5. Hybrid Access Model Performance

quantify the performance gains from HCL's hybrid data access model, we run the following synthetic workload and we measure the performance in MBs per second. Each client process issues 8192 write operations on the target partition. We scale the operation size from 4KB to 8MB and measure the achieved bandwidth. For intra-node access, the target partition resides on the same node as the client processes, whereas, for inter-node, the target partition resides on the server node remotely.

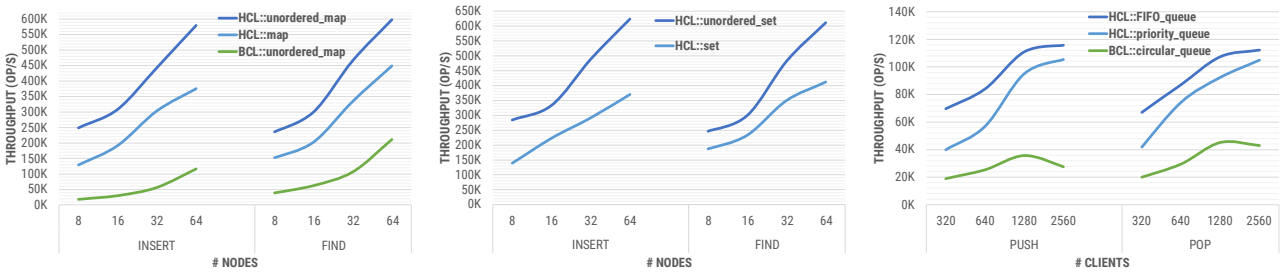
Intra-Node access: The results of the test are shown in Figure 5(a). The memory performance of an Ares node using *Stream* benchmark using 40 threads is roughly 65 GB/sec. We observe in the figure, the bandwidth achieved by HCL is significantly higher than BCL. Specifically, HCL is 2x to 20x (from 512 KB and 64 KB respectively) faster on inserts and 1.5x to 7.2x faster on finds. This performance improvement stems from HCL's hybrid data access model where all operations targeting a co-located partition bypass the RPC-over-RDMA infrastructure and use direct shared memory accesses to serve the requests. The average bandwidth BCL achieves is 4GB/s and 12GB/s for inserts and finds respectively. The performance improvement for finds is lower than it is for inserts since BCL finds perform fewer CAS operations than inserts, and, hence, are less expensive. On the other hand, HCL achieves approximately 45GB/s and 55GB/s for inserts and finds respectively at 32KB event size and maintains it thereafter.

Inter-Node access: The results of the test are shown in Figure 5(b). The average network performance between two nodes in Ares cluster is approximately 4.5 GB/s as measured by the OSU network benchmark [37]. The inter-node performance of HCL, when compared to BCL, is 3.1x to 12x faster for inserts and 1.1x to 9x faster for finds. This is due to the differences between imperative and procedural programming that BCL and HCL adopt respectively. As stated earlier, for each operation, BCL needs to perform multiple remote CAS operations whereas HCL bundles them all in one remote call. As a result, BCL reaches a bandwidth of 1.3GB/s for inserts and 4GB/s for finds at 1MB event size. In contrast, HCL reaches similar bandwidth of 4GB/s to 4.2GB/s for both inserts and finds since both operations involve the same amount of work in data transfer. In BCL, the number of CAS operations necessary for finds is smaller compared to inserts, which explains the performance difference.

In both of the above cases, BCL runs out of memory for cases above 1MB, even though there was more than enough memory on the node for the test. This is because client-side operations require exclusive RDMA buffers to avoid corruption. This increases the overall requirement of memory for BCL. Throughout this test, we observe that the overall capacity allocated to BCL should not exceed 60% of the total node memory to ensure successful completion.

C. Distributed Data Structure Scaling Results

To quantify the effectiveness of distributing HCL data structures, we run a synthetic workload that scales both multi-partition DDSs (i.e., maps and sets) and single-partition DDSs (i.e., queues). For maps and sets, 64 client-nodes (i.e., 2560 processes) issue 8192 operations of 64KB size while we scale the number of partitions from 8 to 64 nodes. For queues, we host the queue on one partition and scale the number of clients issuing requests from 320 to 2560 processes. This test measures the throughput in operations per second. We compare HCL's performance with BCL, but only for data structures that are available in both libraries such as unordered_map and circular queue (i.e., sets and ordered data structures are



(a) Maps (b) Sets (c) Queues

Fig. 6. Scaling HCL Data Structures.

not implemented within BCL). Results are shown in Figure 6 where the X-axis plots the scale of the test and the Y-axis the throughput achieved in operations per second.

Maps: Figure 6(a) shows the results of the tests performed with maps. The `HCL::unordered_map` and `HCL::map` scale linearly as we increase the number of partitions. We observe that as the number of partitions increases, operations become more parallel and a higher network saturation is achieved. The `HCL::map` (i.e., ordered map version) is 54% slower than `HCL::unordered_map` due to the difference in operation complexity. Specifically, the cost of insertion and find operations for the `HCL::map` is $O(log(n))$ as opposed to $O(1)$ for the `HCL::unordered_map`. Inserts for `BCL::unordered_map` are approximately 9.1x slower than HCL's whereas finds are roughly 4.5x slower on average. This performance improvement stems from HCL's hybrid data access model that optimizes co-located operations using shared memory. Further, the `HCL::map` outperforms BCL by 5.5x for inserts and 3.1x for finds on average. Note that BCL scales finds better than inserts due to an inherent smaller number of client-side locking of CAS operations. HCL scales both inserts and finds similarly as it uses lock-free data structures.

Sets: Figure 6(b) shows the results of the tests performed with sets. The `HCL::unordered_set` shows close to linear scaling, as the number of partitions increases, and, achieves a throughput of 620K operations/sec for the 64 partition test case. These results are similar to the `HCL::unordered_map` as internally they use the same lock-free hash data structure. However, sets only contain a single key per element, which reduces the serialization cost. Hence, they are 7% to 14% faster than the map counterparts. Similar to the maps, the ordered version of the set (i.e., `HCL::set`) also scales linearly, but performs worse than the unordered version due to the increased complexity of insertions and finds.

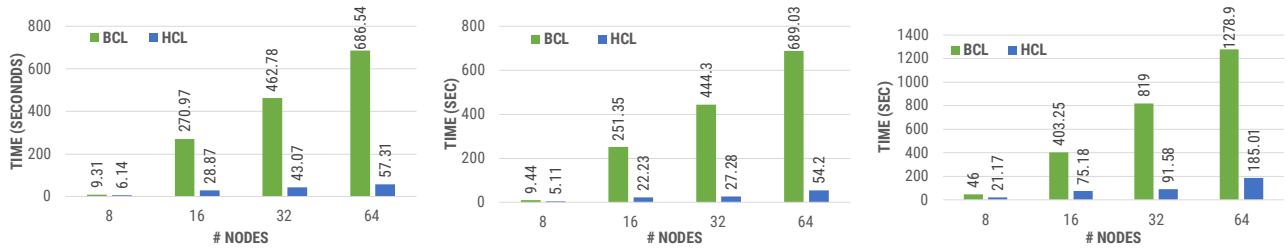
Queues: Figure 6(c) shows the performance results of both FIFO and priority queue. The throughput initially improves as the number of clients increases, but eventually reaches a peak at around 1280 clients (i.e., network is fully saturated). After that, the throughput plateaus since the network experiences congestion and operations are serialized. Additionally, the `HCL::priority_queue` is 30% slower than the `HCL::FIFO_queue` on average because of its logarithmic operation complexity (i.e., $O(log(n))$). BCL achieved a 35K push and 43K pop maximum throughput, significantly lower than HCL. This drop in throughput is caused by BCL's multiple client-side CAS operations on the remote memory (per each push and pop), which incurs additional network cost, and, thus, lowers the throughput. This phenomenon gets exaggerated in the largest scale (i.e., 2560 clients) where the client-side synchronization hurts the overall BCL performance.

D. Real Workloads

To evaluate how HCL handles real-world representative workloads, we run ISx benchmark and Meraculous. We calculate the overall time taken to finish the test and compare the results against BCL. Additionally, we scale the test from 8 nodes to 64 nodes to test the scalability of our solution. Results are shown in Figure 7 where the X-axis shows the number of nodes and Y-axis depicts the time elapsed in seconds. All tests showcase results by weak-scaling (i.e., data increases as the number of nodes increase). Results shown are an average of 5 executions.

1) ISx Benchmark: The ISx is a bucket sort benchmark that sorts uniformly distributed data. It consists of two phases: a data distribution phase and a local sorting phase. In the data distribution phase, processes use pre-existing knowledge about the distribution of the randomly generated data to assign each key to a bucket. By default, there is one bucket on each node. After this stage, each process performs a local sort on its received data and then the processes exchange buckets to get the final sorted list. Figure 7(a) shows the result of running ISx benchmark using HCL and BCL. As can be seen, BCL takes 686 seconds for the biggest scale and scales linearly. In contrast, HCL takes 57 seconds in the largest scale and scales sub-linearly (i.e., around 1.4x as we increase the number of nodes). This performance gain using HCL stems from utilizing a priority queue which sorts the data as it arrives. Hence, each process does not need to perform a push and then a sort separately. This optimization is possible due to the support of a priority queue data structure. Such a priority queue can keep the data sorted in $O(log(n))$ and the cost of sorting gets hidden behind the data movement via the network.

2) Meraculous Genome Assembly: The Meraculous application kernel is a collection of two benchmarks taken from a large-scale scientific application: `contig` generation and `k-mer` counting. The contig generation is a *de novo* genome assembly pipeline that uses an unordered map to traverse a de Bruijn graph of overlapping symbols.



(a) ISx Benchmark (b) Genome Contig Generation Performance (c) Genome k-mer counting Performance

Fig. 7. Performance evaluation with real workloads.

Similarly, k-mer counting uses an unordered map to compute a histogram describing the number of occurrences of each k-mer across reads of a DNA sequence. The performance of the unordered map is critically crucial to the overall performance of this application. More details on how this application uses hashmaps can be found in Brock et al. [11]. In this test, we evaluate HCL for both these cases and compare it with BCL's performance. The results are shown in Figures 7(b) and 7(c). We observe that BCL finishes the contig generation kernel in 9 seconds in the smallest case to 689 seconds in the largest case. On the other hand, HCL finishes the test 1.8x faster in smallest case to 12x in largest case. A similar result is seen in k-mer counting kernel where HCL is 2.17x to 8x faster than BCL. This is due to the fact that HCL's distributed hashmap, as shown before, achieved a better throughput than BCL in all cases. In summary, HCL's implementations of distributed maps have been shown to outperform BCL, mostly due to a paradigm shift from an imperative to a procedural programming one. HCL enhances BCL and significantly boosts performance for parallel applications.

V. Related Work

UPC++ [9], [10] is a C++ library and superset language which is based on the PGAS distributed programming model. UPC++ is in between being a unique programming language and a simple library; it requires a unique compiler, but it is possible to compile it to a legacy C++ code. It focuses on an asynchronous model, much like HCL. Underneath the programming model, everything is expressed as a specialized RPC call (active messages), which are executed atomically. However, this protocol doesn't utilize RDMA but instead uses the CPU core, which decreases its throughput. In HCL, we propose to use an RDMA-based RPC protocol, which can utilize the RDMA infrastructure to provide fast, procedural programming. The current version of UPC++ also lacks properly-implemented high-level abstractions of data structures, which makes it hard to port from the standard library. Finally, UPC++ is tied to the GASNet communication library instead of providing support for multiple backends.

DASH [12] is a C++ library that offers a PGAS programming paradigm. It largely focuses on the structural computational grid with highly efficient support for distributed arrays and matrices. While data structures supported by DASH are generic, they do not support the storing of complex data-types. SHAD [38] is another C++ library which enables complex data structures such as Map, Vector, Sets, and Arrays. However, the support for ordered data structures such as Queues, Priority Queues, Ordered Sets and Ordered Map is unsupported. These data structures enabled distributed transactions, leader election algorithms, and P2P data sharing. STAPL [39] is an STL-like library of parallel algorithms and data structures in C++. STAPL is written at a higher level of abstraction than HCL for special higher-order functions such as a map, reduce and for-each. Finally, Co-array Fortran [40], Hierarchically Tiled Arrays [41], HPX [42], GAM Nets [43], Global Arrays [44], and Multipol [45] are other popular parallel programming libraries which expose concurrent data structures on top of low-level libraries such as MPI. These libraries are highly limited in ds support and do not include complex high-level ds such as hash maps and queues. They mainly focus on simple data structures such as arrays, which can be written or read by multiple processes. By contrast, HCL takes a comprehensive view on exposing data structures in the global address space using RPC over RDMA protocol.

VI. Conclusions

To build parallel programs, scientific applications require high-level abstractions such as distributed maps, sets, and queues. In this paper, we present the Hermes Container Library (HCL) that can boost scientific productivity by providing a high-level and flexible interface. HCL offers high-performance C++ STL-like data containers that can be automatically distributed on many nodes in a cluster. HCL introduces a novel procedural programming approach, via an enhanced RPC-over-RDMA protocol, to perform operations on remote memory segments. This approach is shown to be effective and flexible for asynchronous communication patterns. We introduce HCL's DataBox abstraction that offers several features including persistence, complex data types, hybrid access model, etc. In this work, we build upon BCL, a cross-platform distributed data structures library, while extending both performance and functionality. Evaluation results show a 2x to 12x boost in performance over BCL for a collection of different workloads.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant no. OCI-1835764 and CSR-1814872.

REFERENCES

[1] A. Y. Lokhov, M. Mézard, H. Ohta, and L. Zdeborová, "Inferring the origin of an epidemic with a dynamic message-passing algorithm," *Physical Review E*, vol. 90, no. 1, p. 012801, 2014.

[2] A. Bhatelé, E. Bohm, and L. V. Kalé, "Optimizing communication for charm++ applications by reducing network contention," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, 2011.

[3] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010, p. 2.

[4] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker, *MPI–the Complete Reference: the MPI core*. MIT press, 1998, vol. 1.

[5] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan *et al.*, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1422–1434, 2018.

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[7] M. Weiland, "Chapel, fortress and x10: novel languages for hpc," *EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706*, vol. 1, Oct 2007. [Online]. Available: https://tinyurl.com/rxyr3sx

[8] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella *et al.*, "Titanium: a high-performance java dialect," *Concurrency and Computation: Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.

[9] J. Bachan, D. Bonachea, P. H. Hargrove, S. Hofmeyr, M. Jacquelin, A. Kamil, B. van Straalen, and S. B. Baden, "The upc++ pgas library for exascale computing," in *Proceedings of the Second Annual PGAS Applications Workshop*. ACM, 2017, p. 7.

[10] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "Upc++: a pgas extension for c++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1105–1114.

[11] B. Brock, A. Bulu, and K. Yelick, "Bcl: A cross-platform distributed data structures library," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: ACM, 2019, pp. 102:1–102:10. [Online]. Available: http://doi.acm.org/10.1145/3337821.3337912

[12] K. Fürlinger, T. Fuchs, and R. Kowalewski, "Dash: A c++ pgas library for distributed data structures and parallel algorithms," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. Ieee, 2016, pp. 983–990.

[13] M. P. I. Forum, *MPI: a message passing interface standard: version 2.1; Message Passing Interface Forum, June 23, 2008*. University of Tennessee, 2008.

[14] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu, "Locality-centric thread scheduling for bulk-synchronous programming models on cpu architectures," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 257–268.

[15] P. W. Frey and G. Alonso, "Minimizing the hidden cost of rdma," in *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 2009, pp. 553–560.

[16] D. Y. Yoon, M. Chowdhury, and B. Mozafari, "Distributed lock management with rdma: decentralization without starvation," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1571–1586.

[17] W. D. Gropp and R. Thakur, "Revealing the performance of mpi rma implementations," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2007, pp. 272–280.

[18] "fortress-spec.pdf," http://www.ccs.neu.edu/home/samth/fortress-spec.pdf, accessed: 2020-03-21.

[19] "Internet parallel computing archive : Standards : Hpf," http://wotug.org/parallel/standards/hpf/, accessed: 2020-03-21.

[20] B. L. Chamberlain, S.-E. Choi, C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "Zpl: A machine independent programming language for parallel computers," *IEEE Transactions on Software Engineering*, vol. 26, no. 3, pp. 197–211, 2000.

[21] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using rdma and htm," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 87–104.

[22] P. Stuedi, B. Metzler, and A. Trivedi, "jverbs: Rdma support for java®," *IBM Research Library*, p. 11, 2016.

[23] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[24] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 401–414.

[25] G. Bosilca, G. Fedak, O. Richard, and F. Cappello, "High performance computing with rpc programming style," in *Proceedings of the First Myrinet User Group Conference MUG*, 2000.

[26] Mellanox, "Rdma aware programming user manual," 5 2015. [Online]. Available: https://tinyurl.com/ycpaydw3

[27] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–8.

[28] K. Maeda, "Comparative survey of object serialization techniques and the programming supports," *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 5, no. 12, pp. 1488–1493, 2011.

[29] B. Schäling, *The boost C++ libraries*. Boris Schäling, 2011.

[30] N. Nguyen and P. Tsigas, "Lock-free cuckoo hashing," in *2014 IEEE 34th international conference on distributed computing systems*. IEEE, 2014, pp. 627–636.

[31] A. Natarajan, L. H. Savoie, and N. Mittal, "Concurrent wait-free red black trees," in *Symposium on Self-Stabilizing Systems*. Springer, 2013, pp. 45–60.

[32] E. Ladan-Mozes and N. Shavit, "An optimistic approach to lock-free fifo queues," in *International Symposium on Distributed Computing*. Springer, 2004, pp. 117–131.

[33] D. Zhang and D. Dechev, "A lock-free priority queue design based on multi-dimensional linked lists," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 613–626, 2015.

[34] U. Hanebutte and J. Hemstad, "Isx: a scalable integer sort for co-design in the exascale era," in *2015 9th International Conference on Partitioned Global Address Space Programming Models*. IEEE, 2015, pp. 102–104.

[35] J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, and D. S. Rokhsar, "Meraculous: de novo genome assembly with short paired-end reads," *PloS one*, vol. 6, no. 8, 2011.

[36] D. Olasoji, L. Yingqi, K. Eric, and G. Agata, "Performance analysis tool," 6 2015. [Online]. Available: https://github.com/intel-hadoop/PAT

[37] V. Plugaru, "Ul hpc mpi tutorial," 8 2017.

[38] V. G. Castellana and M. Minutoli, "Shad: The scalable high-performance algorithms and data-structures library," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 442–451.

[39] G. Tanase, A. Buss, A. Fidel, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu *et al.*, *The STAPL parallel container framework*. ACM, 2011, vol. 46, no. 8.

[40] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," in *ACM Sigplan Fortran Forum*, vol. 17, no. 2. ACM, 1998, pp. 1–31.

[41] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. Von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2006, pp. 48–57.

[42] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[43] M. Drocco, "Parallel programming with global asynchronous memory: Models, c++ apis and implementations," 2017.

[44] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[45] S. Chakrabarti, E. Deprit, E.-J. Im, J. Jones, A. Krishnamurthy, C.-P. Wen, and K. Yelick, "Multipol: A distributed data structure library," in *Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 1995.