BCL: A Cross-Platform Distributed Data Structures Library

Benjamin Brock, Aydın Buluç, Katherine Yelick
University of California, Berkeley
Lawrence Berkeley National Laboratory
{brock,abuluc,yelick}@cs.berkeley.edu

ABSTRACT

One-sided communication is a useful paradigm for irregular parallel applications, but most one-sided programming environments, including MPI's one-sided interface and PGAS programming languages, lack application-level libraries to support these applications. We present the Berkeley Container Library, a set of generic, cross-platform, high-performance data structures for irregular applications, including queues, hash tables, Bloom filters and more. BCL is written in C++ using an internal DSL called the BCL Core that provides one-sided communication primitives such as remote get and remote put operations. The BCL Core has backends for MPI, OpenSHMEM, GASNet-EX, and UPC++, allowing BCL data structures to be used natively in programs written using any of these programming environments. Along with our internal DSL, we present the BCL ObjectContainer abstraction, which allows BCL data structures to transparently serialize complex data types while maintaining efficiency for primitive types. We also introduce the set of BCL data structures and evaluate their performance across a number of high-performance computing systems, demonstrating that BCL programs are competitive with hand-optimized code, even while hiding many of the underlying details of message aggregation, serialization, and synchronization.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**.

KEYWORDS

Parallel Programming Libraries, RDMA, Distributed Data Structures

ACM Reference Format:
Benjamin Brock, Aydın Buluç, Katherine Yelick. 2019. BCL: A Cross-Platform Distributed Data Structures Library. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337912

1 INTRODUCTION

Writing parallel programs for supercomputers is notoriously difficult, particularly when they have irregular control flow and complex data distribution; however, high-level languages and libraries can make this easier. A number of languages have been developed for high-performance computing, including several using the *Partitioned Global Address Space (PGAS)* model: Titanium, UPC, Coarray Fortran, X10, and Chapel [9, 11, 12, 25, 29, 30]. These languages are especially well-suited to problems that require asynchronous one-sided communication, or communication that takes place without a matching receive operation or outside of a global collective. However, PGAS languages lack the kind of high level libraries that exist in other popular programming environments. For example, high-performance scientific simulations written in MPI can leverage a broad set of numerical libraries for dense or sparse matrices, or for structured, unstructured, or adaptive meshes. PGAS languages can sometimes use those numerical libraries, but are missing the data structures that are important in some of the most irregular parallel programs.

This paper describes the Berkeley Container Library (BCL) that is intended to support applications with irregular patterns of communication and computation and data structures with asynchronous access, for example hash tables and queues, that can be distributed across processes but manipulated independently by each process. BCL is designed to provide a complementary set of abstractions for data analytics problems, various types of search algorithms, and other applications that do not easily fit a bulk-synchronous model. BCL is written in C++ and its data structures are designed to be *coordination free*, using one-sided communication primitives that can be executed using RDMA hardware without requiring coordination with remote CPUs. In this way, BCL is consistent with the spirit of PGAS languages, but provides higher level operations such as *insert* and *find* in a hash table, rather than low-level remote read and write. As in PGAS languages, BCL data structures live in a global address space and can be accessed by every process in a parallel program. BCL data structures are also *partitioned* to ensure good locality whenever possible and allow for scalable implementations across multiple nodes with physically disjoint memory.

BCL is cross-platform, and is designed to be agnostic about the underlying communication layer as long as it provides one-sided communication primitives. It runs on top of MPI's one-sided communication primitives, OpenSHMEM, and GASNet-EX, all of which provide direct access to low-level remote read and write primitives to buffers in memory [6, 10, 16]. BCL provides higher level abstractions than these communication layers, hiding many of the details of buffering, aggregation, and synchronization from users that are specific to a given data structure. BCL also has an experimental UPC++ backend, allowing BCL data structures to be used inside another high-level programming environment. BCL uses a high-level data serialization abstraction called ObjectContainers to allow the storage of arbitrarily complex datatypes inside BCL data structures. BCL ObjectContainers use C++ compile-time type introspection to avoid introducing any overhead in the common case that types are byte-copyable.

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337912

We present the design of BCL with an initial set of data structures and operations. We then evaluate BCL's performance on ISx, an integer sorting mini-application, Meraculous, a mini-application taken from a large-scale genomics application, and a collection of microbenchmarks examining the performance of individual data structure operations. We explain how BCL's data structures and design decisions make developing high-performance implementations of these benchmarks more straightforward and demonstrate that BCL is able to match or exceed the performance of both specialized, expert-tuned implementations as well as general libraries across three different HPC systems.

1.1 Contributions

(1) A distributed data structures library that is designed for high performance and portability by using a small set of core primitives that can be executed on four distributed memory backends
(2) The BCL ObjectContainer abstraction, which allows data structures to transparently handle serialization of complex types while maintaining high performance for simple types
(3) A distributed hash table implementation that supports fast insertion and lookup phases, dynamic message aggregation, and individual insert and find operations
(4) A distributed queue abstraction for many-to-many data exchanges performed without global synchronization
(5) A distributed Bloom filter which achieves fully atomic insertions using only one-sided operations
(6) A collection of distributed data structures that offer *variable levels of atomicity* depending on the call context using an abstraction called concurrency promises
(7) A fast and portable implementation of the Meraculous benchmark built in BCL
(8) An experimental analysis of irregular data structures across three different computing systems along with comparisons between BCL and other standard implementations.

2 BACKGROUND AND HIGH-LEVEL DESIGN

Several approaches have been used to address programmability issues in high-performance computing, including parallel languages like Chapel, template metaprogramming libraries like UPC++, and embedded DSLs like STAPL. These environments provide core language abstractions that can boost productivity, and some of them have sophisticated support for multidimensional arrays. However, none of these environments feature the kind of rich data structure libraries that exist in sequential programming environments like C++ or Java. A particular need is for distributed memory data structures that allow for nontrivial forms of concurrent access that go beyond partitioned arrays in order to address the needs of irregular applications. These data structures tend to have more complicated concurrency control and locality optimizations that go beyond tiling and ghost regions.

Our goal is to build robust, reusable, high-level components to support these irregular computational patterns while maintaining performance close to hardware limits. We aim to achieve this goal using the following design principles.

Low Cost for Abstraction. While BCL offers data structures with high-level primitives like hash table and queue insertions, these commands will be compiled directly into a small number of one-sided remote memory operations. Where hardware support is available, all primary data structure operations, such as reads, writes, inserts, and finds, are executed purely in RDMA *without requiring coordination with remote CPUs*.

Portability. BCL is cross-platform and can be used natively in programs written in MPI, OpenSHMEM, GASNet-EX, and UPC++. When programs only use BCL data structures, users can pick whichever backend's implementation is most optimized for their system and network hardware.

Software Toolchain Complexity. BCL is a *header-only* library, so users need only include the appropriate header files and compile with a C++-14 compliant compiler to build a BCL program. BCL data structures can be used in part of an application without having to re-write the whole application or include any new dependencies.

3 BCL CORE

The BCL Core is the cross-platform internal DSL we use to implement BCL data structures. It provides a high-level PGAS memory model based on global pointers, which are C++ objects that allow the manipulation of remote memory. Similar to other PGAS programming models, each process has a *shared memory segment*, and each process can allocate memory in that segment using global pointers, which in BCL are regular C++ objects that can be passed around between processes or stored in global memory. Global pointers support remote get and remote write operations. Remote completion of put operations is not guaranteed until after a memory fence such as a flush or a barrier.

Although BCL is not designed for bulk synchronous programming, it provides a limited set of collective operations such as broadcast and allreduce for transporting pointer and control values.

BCL adheres firmly to the idea of one-sided communication and avoids the use of remote operations that require the use of a remote CPU. The BCL core instead relies on remote memory operations and atomics, which can be supported by network hardware and do not interrupt computations running on the CPU. BCL backends must implement at least the atomic compare-and-swap (CAS) operation, since all other atomic memory operations (AMOs) can be implemented on top of CAS [19]. Other common atomics include fetch-and-op atomics which can perform addition and bitwise operations. More details on the semantics of the BCL Core are in our preprint [7].

Backends, which include MPI, OpenSHMEM, GASNet-EX, and an in-progress UPC++ backend, provide provide a small number of functions to support the BCL Core. Necessary functions include an init function that allocates symmetric shared memory segments, `barrier`, `read`, and `write` functions, and at least an atomic CAS operation.

4 PARALLEL PATTERNS IN BCL

When choosing data structures to implement in BCL, we wanted to focus on data structures that could exploit particular high-level parallel patterns [22, 23]. While BCL also efficiently supports commonly known data structure patterns such as the Distributed Array Pattern [22], its novelty lies in its support for more challenging irregular data access patterns as first-class citizens. In particular, we chose to focus on exposing high-level data structures that exploit two parallel patterns: (1) fine-grained, low-latency reads and writes, and (2) asynchronous many-to-many redistribution of data. These patterns occur in many applications that perform concurrent reads and writes in an unpredictable manner, with prime examples in graph algorithms, computational chemistry, and bioinformatics. These patterns can also be used in loosely synchronous applications that require data redistribution due to changes in the computational structure as the algorithms proceed [26].

4.1 Fine-Grained RDMA Operations

For the first pattern, we wanted to provide high-level interfaces for fine-grained operations that are potentially complex, such as hash table operations, but in many cases will be executed as a single RDMA operation. For these low-latency operations, designing a low-cost, header-only library where user code is compiled down to a small number of calls to a backend library is essential to achieve performance. Also essential to achieving performance for low-latency operations across a variety of computing platforms is supporting multiple backends, since oftentimes the best communication backend varies across supercomputing platforms. Examples of data structures we implemented which expose this pattern include hash tables and Bloom filters, discussed in Sections 5.2 and 5.4.

4.2 Many-to-Many Data Redistribution

For the second pattern, we are interested in applications where each process wishes to push data to other processes in an asynchronous, arbitrary manner. MPI all-to-all provides a restricted implementation of this pattern, where each process gathers its data to be sent to each other process, then all processes take part in a bulk synchronous all-to-all operation. While there are asynchronous versions of MPI all-to-all, it still restricts processes from generating new data after the all-to-all operation has started, thus limiting the possibility for overlap between communication and computation. Sometimes this pattern is explicitly present, such as in sorting or histogramming, but sometimes it can be exposed by buffering and aggregating fine-grained operations. In this paper, we first build queue data structures (Section 5.1) that allow for arbitrary data redistribution using asynchronous queue insertions. Then, we design a “hash table buffer” data structure (Section 5.3) that allows users to buffer and aggregate hash table insertions transparently, transforming fine-grained, latency-bound operations into bulk, bandwidth-bound ones.

5 BCL DATA STRUCTURES

BCL data structures are split into two categories: *distributed* and *hosted*. Distributed data structures live in globally addressable memory and are automatically distributed among all the ranks in a BCL program. Hosted data structures, while resident in globally addressable memory, are hosted only on a particular process. All other processes may read or write from the data structure lying on the host process. We have found hosted data structures to be an important building block in creating distributed data structures.

Data Structure	Locality	Description
`BCL::HashMap`	Distributed	Hash Table
`BCL::CircularQueue`	Hosted	Multiple Reader/Writer Queue
`BCL::FastQueue`	Hosted	Multi-Reader *or* Multi-Writer Queue
`BCL::HashMapBuffer`	Distributed	Aggregate hash table insertions
`BCL::BloomFilter`	Distributed	Distributed Bloom filter
`BCL::DArray`	Distributed	1-D Array
`BCL::Array`	Hosted	1-D Array on one process

Table 1: A summary of BCL data structures.

All BCL data structures are *coordination free*, by which we mean that primary data structure operations, such as insertions, deletions, updates, reads, and writes, can be performed without coordinating with the CPUs of other nodes, but purely in RDMA where hardware support is available. Other operations, such as resizing or migrating hosted data structures from one node to another, may require coordination. In particular, operations which modify the size and location of the data portions of BCL data structures must be performed collectively, on both distributed and hosted data structures. This is because coordination-free data structure methods, such as insertions, use global knowledge of the size and location of the data portion of the data structure. For example, one process cannot change the size or location of a hash table without alerting other processes, since they may try to insert into the old hash table memory locations. Tables 1 and 2 give an overview of the available data structures and operations. Table 2 also gives the best-case cost of each operation in terms of remote reads R, remote writes W, atomic operations A, local operations ℓ, and global barriers B. As demonstrated by the table, each high-level data structure operation is compiled down to a small number of remote memory operations.

All BCL data structures are also generic, meaning they can be used to hold any type, including complex, user-defined types. Most common types will be handled automatically, without any intervention by the user. See Section 6 for a detailed description of BCL’s lightweight serialization mechanism.

Many distributed data structure operations have multiple possible implementations that offer varying levels of atomicity. Depending on the context of a particular callsite, only some of these implementations may be valid. We formalize a mechanism, called *concurrency promises*, that allows users to optionally assert invariants about a callsite context. This allows BCL data structures to use optimized implementations that offer fewer atomicity guarantees when a user guarantees that this is possible. This mechanism is discussed in Section 7.

5.1 Queues

BCL includes two types of queues: one, `CircularQueue`, is a general multi-reader, multi-writer queue which supports variable levels of atomicity. The second, `FastQueue`, supports multiple readers *or* multiple writers, but requires that read and write phases be separated by a barrier. Both queues are implemented as ring buffers and are initialized with a fixed size as a *hosted* data structure, so while a queue is globally visible, it is resident on only one process at a time.

FastQueue uses three shared objects: a data segment, where queue elements are stored; a shared integer that stores the head of the queue; and a shared integer that stores the tail of the queue. To insert, a process first increments the tail using an atomic fetch-and-add operation, checks that this does not surpass the head pointer, and then inserts its value or values into the data segment of the

Data Structure	Method	Collective	Description	Cost
BCL::HashMap				
	`bool insert(const K &key, const V &val)`	N	Insert item into hash table.	2A + W
	`bool find(const K &key, V &val)`	N	Find item in table, return val.	2A + R
BCL::BloomFilter				
	`bool insert(const T &val)`	N	Insert item into Bloom filter, return true if already present.	A
	`bool find(const T &val)`	N	Return true if item is likely to be in filter, false otherwise.	R
BCL::CircularQueue				
	`bool push(const T &val)`	N	Insert item into queue.	$2A + W$
	`bool pop(T &val)`	N	Pop item into queue.	$2A + R$
	`bool push(const std::vector <T> &vals)`	N	Insert items into queue.	$2A + nW$
	`bool pop(std::vector <T> &vals, size_t n)`	N	Pop items from queue.	$2A + nR$
	`bool local_nonatomic_pop(T &val)`	N	Nonatomically pop item from a local queue.	ℓ
	`void resize(size_t n)`	Y	Resize queue.	$B + \ell$
	`void migrate(size_t n)`	Y	Migrate queue to new host.	$B + nW$

Table 2: A selection of BCL data structure methods. Costs are best case, without any concurrency promises. R, W, A, B, ℓ, and n are the costs of a remote read, write, atomic memory op., barrier, local memory op., and number of elements, respectively.

Method	Concurrency Promise	Description	Cost
insert			
(a)	`find | insert`	Fully Atomic	$2A + W$
(b)	`local`	Local Insert	ℓ
find			
(c)	`find | insert`	Fully Atomic	$2A + R$
(d)	`find`	Only Finds	R

Table 3: Implementations for hash table methods.

Method	Concurrency Promise	Description	Cost
push			
(a)	`push | pop`	Fully Atomic	$2A + W$
(b)	`push`	Only Pushes	$2A + W$
(c)	`local`	Local Push	ℓ
pop			
(d)	`push | pop`	Fully Atomic	$2A + R$
(e)	`pop`	Only Pops	$2A + R$
(f)	`local`	Local Pop	ℓ

Table 4: Implementations for circular queue methods.

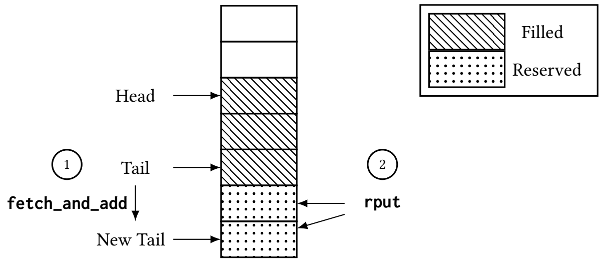

Figure 1: Process for pushing values to a BCL `FastQueue`. First (1) a `fetch_and_add` operation is performed, which returns a reserved location where values can be inserted. Then (2) the values to be inserted are copied to the queue.

queue. An illustration of a push operation is shown in Figure 1. In general, the head overrun check is performed without a remote memory operation by caching the position of the head pointer, so an insertion requires only two remote memory operations. We similarly cache the location of the tail pointer, so pops to the queue usually require only one atomic memory operation to increment the head pointer and one remote memory operation to read the popped values.

CircularQueue. To support concurrent reads and writes, circular queue has an additional set of head and tail pointers which indicate which portions of data in the queue are ready to be read. There are multiple implementations of push and pop for a circular queue data structure, as listed in Table 4.

Push and Pop Operations. The default fully atomic implementation used for insertion (Table 4a) into a circularqueue data structure involves 2 atomic operations and a remote put operation with a flush. First, we issue a fetch-and-add operation to increment the tail pointer, then write the data to the queue and flush it. Finally, we must perform a CAS operation to increment the "tail ready" pointer, indicating that the pushed data is ready to be read. A CAS is necessary for the final step because a fetch-and-add could increment the ready pointer to mistakenly mark other processes' writes as ready to be read. In the case where no pop operations will be performed before a barrier, we may perform the final atomic increment using a fetch-and-add (Table 4b). An analogous implementation is used for pop operations (Table 4d and 4e).

Both queues support resizing as well as migrating to another host process, both as collective operations. We evaluate the performance of our circular queue data structures in Section 8.1.

Advantage of FastQueue. `FastQueue` has the advantage of requiring one fewer AMO per push or pop. While the `CircularQueue` does support variable levels of atomicity, allowing the final pop to be a single non-blocking fetch-and-add operation, we felt that this was an important enough overhead to warrant a separate version of the data structure, since queues that support only multi-reader and multi-writer *phases* are crucial to several of the algorithms that we explored.

5.2 Hash Table

BCL's hash table is implemented as a single logically contiguous array of hash table buckets distributed block-wise among all processes. Each bucket is a struct including a key, value, and status flag. Our hash table uses open addressing with quadratic probing to resolve hash collisions. As a result, neither insert nor find operations to our hash table require any coordination with remote ranks. Where hardware support is available, hash table operations will only use RDMA operations.

Interface. BCL's `BCL::HashMap` is a distributed data structure. Users can create a `BCL::HashMap` by calling the constructor as a collective operation. BCL hash tables are created with a fixed key and value type as well as a fixed size. BCL hash tables use ObjectContainers, discussed in Section 6, to store any arbitrary data types. The hash table supports two primary methods, `insert` and `find`. Section 8 gives a performance analysis of our hash table.

Atomicity. By default, hash table insert and find operations are atomic with respect to one another, including simultaneous insert operations and find operations using the same key. In addition to this default level of atomicity, users can pass a concurrency promise as an optional argument at each callsite that can allow the data structure to select a more optimized implementation with less strict atomicity guarantees. All the available implementations for insert and find operations are shown in Table 3.

Our hash table uses a lightweight, per-bucket locking scheme. Each hash table bucket has a 32-bit *used* flag that ensures atomicity of operations. The lowest 2 bits of this flag indicate the reservation status of the bucket. There are three possible states: (1) free, (2) reserved, and (3) ready. The free state represents an unused bucket, the reserved state represents a bucket that has been reserved is being modified, and the ready state indicates that a bucket is ready to be read. The remaining 30 bits are *read flag* bits, and they indicate, if flipped, that a process is currently reading the hash table entry. This prevents another process from writing to the entry before the other process has finished reading.

Insert Operations. The default, fully atomic process for inserting (Table 3a) requires two atomic memory operations (AMOs) and a remote put with a flush. First, the inserting process computes the appropriate bucket. Then it uses a compare-and-swap (CAS) operation to set the bucket's status to reserved, a remote put to write the correct key and value to the reserved bucket, followed by a flush to ensure completion of the put, then finally an atomic XOR to set the status of the bucket to ready.

In some special cases, we may wish to have processes perform local insertions into their own portions of the hash table. This may be done with only local CPU instructions, not involving the NIC. Crucially, this cannot be done when other operations, such as general find or insert operations, might be executed, since CPU atomics are not atomic with respect to NIC atomics. This implementation requires the concurrency promise `local` (Table 3b).

Find Operations. The default, fully atomic implementation of the find operation (Table 3c) again involves two AMOs and a remote read. First, the process uses a fetch-and-or to set a random read bit. This keeps other processes from writing to the hash bucket before the process has finished reading it. Then, it reads the value, and, after reading, unsets the read bit.

In the common case of a *traversal phase* of an application, where no insert operations may occur concurrent with find operations, we may use an alternate implementation that requires no atomic operations (Table 3d), but just a single read operation to read the whole bucket including the reserved flag, key, and value.

Hash Table Size. A current limitation of BCL is that, since hash tables are initialized to a fixed size and do not dynamically resize, an insertion may fail. In the future, we plan to support a dynamically resizing hash table. Currently, the user must call the collective `resize` method herself when the hash table becomes full.

5.3 Buffering Hash Table Insertions

Many applications, such as the Meraculous benchmark, exhibit *phasal* behavior, where there is an insert phase, followed by a barrier, followed by a read phase. We anticipate that this is likely to be a common case, and so have created a hash table *buffer* data structure that accelerates hash table insertion phases. An application programmer can create a new `BCL::HashMapBuffer` on top of an existing hash table. The user then inserts directly into the hash map buffer object using the same methods provided by the hash table. This simple code transformation is demonstrated in Figure 3. While the hash table interface ensures ordering of hash table insertions, insertions into the hash table buffer are non-blocking, and ordering is no longer guaranteed until after an explicit flush operation. The hash table buffer implementation creates a `FastQueue` on each node as well as local buffers for each other node. When a user inserts into the hash table buffer, the insert will be stored in a buffer until the buffer reaches its maximum size, when it will be pushed to the queue lying on the appropriate node to be staged for insertion. At the end of an insert phase, the user calls the `flush()` method to force all buffered insertions to complete. Insertions into the actual table will be completed using a local, fast hash table insertion (Table 3b). The hash map buffer results in a significant performance boost for phasal applications, as discussed in Section 8.2.

5.4 Bloom Filters

A Bloom filter is a space-efficient, probabilistic data structure that answers queries about set membership [5]. Bloom filters can be used to improve the efficiency of hash tables, sets, and other key-based data structures. Bloom filters support two operations, *insert* and *find*. To insert a value into the Bloom filter, we use k hash functions to pick k locations in a bit array that will all be set to one. To find if a value is present in a Bloom filter, we check if each of the corresponding k bits is set, and if so, the value is said to be present. Because of hash collisions, a Bloom filter may return false positives, although it will never return false negatives.

Distributed Bloom Filter. We implement a distributed Bloom filter as a distributed collection of blocked Bloom filters [27], each of which is 64 bits. To insert an element into the distributed Bloom filter, we hash the value once, to pick a Bloom filter, then k times to pick which bits in the filter to set. This allows us to insert into the distributed Bloom filter with a single atomic fetch-and-or operation, which also atomically returns whether the value was previously present. A find operation is completed with a single read operation. More details of our Bloom filter is in our extended preprint [7].

6 BCL OBJECTCONTAINERS

All BCL data structures use BCL ObjectContainers, which provide a transparent abstraction for storing complex data types in distributed memory with low overhead. BCL ObjectContainers are necessary because not all data types can be stored in distributed memory by

```
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();
}
```

Figure 2: Our bucket sort implementation in BCL for the ISx benchmark.

```
BCL::HashMap<int, int> map(size);
BCL::HashMapBuffer<int, int> buffer(map, queue_size,
                                    message_size);
for (...) {
 buffer.insert(key, value);
}
buffer.flush();
```

Figure 3: A small change to user code—inserting into the `HashMapBuffer` instead of the `HashMap`—causes inserts to be batched together.

byte copying. The common case for this is a struct or class, such as the C++ standard library's `std::string`, which contains a pointer. The pointer contained inside the class is no longer meaningful once transferred to another node, since it refers to local memory that is now inaccessible, so we must use some other method to serialize and deserialize our object in a way that is meaningful to remote processes. At the same time, we would like to optimize for the common case where objects *can* be byte copied and avoid making unnecessary copies.

Implementation. BCL ObjectContainers are implemented using the C++ type system. A BCL ObjectContainer is a C++ struct that takes two template parameters: (1) a type of object that the ObjectContainer will hold, and (2) a C++ struct with methods to serialize and deserialize objects of that type. BCL ObjectContainers themselves are of a fixed size and can be byte copied to and from shared memory. An ObjectContainer has a *set* method, which allows the user to store an object in the ObjectContainer, and a *get* method, which allows the user to retrieve the object from the container.

BCL automatically detects and handles trivially serializable types, which do not require serialization, using C++ type traits, and BCL includes automatic handling for a number of common C++ types. Users will usually not have to write their own serialization and deserialization methods unless they wish to use custom types which use heap memory or other local resources.

A finer point of BCL serialization structs is that they may serialize objects to either *fixed length* or *variable length* types. This is handled automatically at compile time by looking at the return type of the serialization struct: if the serialization struct returns an object of any normal type, then the serialized object is taken to be fixed size and is stored directly as a member variable of the serialization struct. If, however, the serialization struct returns an object of the special type `BCL::serial_ptr`, this signifies that the object is *variable length*, even when serialized, so we must instead store a global pointer to the serialized object inside the ObjectContainer.

User-Defined Types. To store user-defined types in BCL data structures, users can simply define serialization structs for their type and inject the struct into the BCL namespace. For byte-copyable types, this struct can be an empty struct that inherits from an "identity serialization" struct.

Copy Elision Optimization. An important consideration when using serialization is overhead in the common case, when no serialization is actually required. In the common byte-copyable case, where the serialization struct simply returns a reference to the original object, intelligent compilers are able to offer some *implicit* copy elision automatically. We have observed, by examining the assembly produced, that the GNU and Clang compilers are able to optimize away unnecessary copies when a ObjectContainer object is retrieved from distributed memory and `get()` is called to retrieve the item lying inside. However, when an array of items is retrieved from distributed memory and unpacked, the necessary loop complicates analysis and prevents the compiler from performing copy elision.

For this reason, BCL data structures perform *explicit* copy elision when reading or writing from an array of ObjectContainers stored in distributed memory when the ObjectContainer inherits from the "identity serialization" struct, which signifies that it is byte copyable. This is a compile-time check, so there is no runtime cost for this optimization.

7 CONCURRENCY PROMISES

As we have shown for various BCL data structures, distributed data structure operations often have multiple alternate implementations, only some of which will be correct in the context in which an operation is issued. A common example of this is *phasal* operations, where data structures are manipulated in separate phases, each of which is separated by a barrier. Figures 2 and 3 both demonstrate phasal operations. Crucially, the barriers associated with phasal operations provide atomicity between different types of operations that often allows the use of implementations with fewer atomicity guarantees. For example, a find operation in a hash table can be executed with a more optimized implementation—a single remote get operation, rather than 2 AMOs and a remote get—when we are guaranteed that only find operations will be executed in the same barrier region.

To allow users to take advantage of these optimized implementations in a straightforward manner, we allow users to optionally

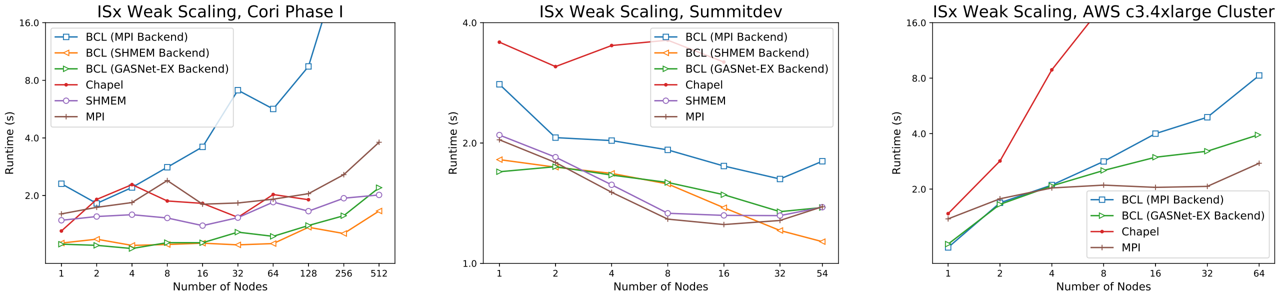

Figure 4: The ISx benchmark on three different computing systems. All runs measure weak scaling with 2^{24} items per process.

Name	Processors	Interconnect
Cori Phase I	Intel Xeon Haswell	Cray Aries
Summitdev	IBM POWER8	Mellanox EDR Infiniband
AWS c3.4xlarge	Intel Xeon	10Gb/s Ethernet

Table 5: Summary of systems used in evaluation.

provide *concurrency promises*, which are lists of data structure operations that could take place concurrently with the operation being issued. To use an optimized version of hash table `find`, we can pass as an extra argument to the `find` function the value `ConProm::HashMap::find`. This indicates that only `find` operations may occur simultaneously with this invocation. Similarly, if in a particular context a `find` operation might also occur concurrently with a `insert` operation, we can pass the concurrency promise `ConProm::HashMap::find | ConProm::HashMap::insert`.

It's important to note that, since C++ template metaprogramming does not have full-program knowledge (unless the whole program is expressed as a single expression), it is not possible to automatically identify these optimization opportunities using a library. Instead, it would require static analysis using either a preprocessing tool or a separate parallel programming language with an intermediate representation that preserves semantic knowledge of data structure operations. Our approach here is to make it easy for the user to provide the invariants, rather than to identify them automatically.

8 EXPERIMENTAL EVALUATION

We evaluated the performance of BCL's data structures using ISx, an integer sorting mini-application, Meraculous, a mini-application taken from large-scale genome assembly, and a collection of microbenchmarks. In order to evaluate the performance portability of BCL programs, we tested the first two benchmarks across three different computer systems, as outlined in Table 5. On Cori, experiments are performed up to 512 nodes. On Summitdev, experiments are performed up to 54 nodes, which is the size of the whole cluster. On AWS, we provisioned a 64 node cluster and performed scaling experiments up to its full size. For reasons of space, the microbenchmarks are presented only on Cori up to 64 nodes.

8.1 ISx Benchmark

We tested our queue's performance by implementing the ISx bucket sort benchmark [17]. ISx is performed on uniformly distributed data and consists of two stages, a distribution stage and a local sort stage. In the distribution stage, processes use pre-existing knowledge about the distribution of the randomly generated data to transfer each key to a bucket, with one bucket on each node by default. Next, each process performs a local sort on its data. The original ISx benchmark includes an MPI implementation, which uses an all-to-all collective for the distribution stage, and an OpenSHMEM implementation, which sends data asynchronously. An implementation in Chapel, a high-level parallel programming language, has also been published [1, 18].

We implemented our bucket sort in BCL using the circular queue data structure. First, during initialization, we place one circular queue on each process. During the distribution phase, each process pushes its keys into the appropriate remote queues. After a global barrier, each node performs a local sort on the items in its queue. During the distribution phase, we perform *aggregation* of inserts to amortize the latency costs of individual inserts. Instead of directly pushing individual items to the remote queues, we first place items in local buffers corresponding to the appropriate remote queue. Once a bucket reaches a set message size, we push the whole bucket of items at once and clear the local bucket. It's important to note that this push is *asynchronous*, meaning that the communication involved with pushing items to the queue can be overlapped with computation involved with sorting the items. The fact that BCL circular queue's push method accepts a vector of items to insert simultaneously makes adding aggregation very straightforward. Even with this optimization, our full BCL sorting benchmark code, including initialization and timing, is only 72 lines long, compared to the original MPI and SHMEM reference implementations at 838 and 899 lines, and the Chapel implementation at 244 lines. A slightly abbreviated version of our implementation is listed in Figure 2.

As shown in Figure 4, our BCL implementation of ISx performs competitively with the reference and Chapel implementations. On Cori, BCL outperforms the other implementations. This is because BCL is able to overlap communication with computation: asynchronous queue insertions overlap with sorting values into buckets. This is an optimization that would be complex to apply in a low-level MPI or SHMEM implementation, but is straightforward using BCL's high-level interface.

There is an upward trend in the BCL scaling curves toward the high extreme of the graph on Cori. This is because as the number of processes increases, the number of values sent to each process decreases. At 512 nodes with 32 processes per node, each process will send, on average, 1024 values to each other process. With our message size of 1024, on average only one push is sent to each other process, and the potential for communication and computation overlap is much smaller, thus our solution degenerates to the synchronous all-to-all solution, and our performance matches the reference SHMEM implementation. Note that performance with the MPI backend is poor on Cori; we believe this is because the MPI implementation is failing to use hardware atomics.

Performance on Summitdev is similar, except for a slight downward trend in all the scaling lines because of cache effects. As the number of processes increases, the keyspace on each node decreases, and the local sort becomes more cache efficient.

PGAS programs historically perform poorly on Ethernet clusters, since they often rely on fast hardware RDMA support. With our bucket sort, we can increase the message size to amortize the cost of slow atomic operations. While our performance on AWS does not scale as well as the reference MPI implementation, we consider the performance acceptable given that it is a high-level implementation running in an environment traditionally deemed the exclusive domain of message-passing. On the Ethernet network, the GASNet-EX backend using the UDP conduit performs better than the MPI backend, which is using Open MPI.

8.2 Genome Assembly

We evaluated our generic hash table by using it to implement one one stage of a de novo genome assembly pipeline, *contig generation*. During contig generation, many error-prone reads recorded by a DNA sequencer have been condensed into k-mers, which are short error-free strands of DNA that overlap each other by exactly k bases. The goal of contig generation is to process k-mers to produce *contigs*, which are long strands of contiguous DNA [14].

Assembling k-mers into longer strands of DNA involves using a hash table to traverse the de Bruijn graph of overlapping k-mers. This is performed by taking a k-mer, computing the next overlapping k-mer in the sequence, and then looking it up in the hash table. This process is repeated recursively until a k-mer is found which does not match the preceding k-mer or has an invalid base.

A fast implementation for contig generation is relatively simple in a serial program, since using any of a large number of generic hash table libraries will yield high performance. However, things are not so simple in distributed memory. The baseline parallel solution for Meraculous, written in UPC, is nearly 4,000 lines long and includes a large amount of boilerplate C code for operations like reading and writing to memory buffers [2].

The implementation of the contig generation phase is greatly simplified by the availability of a generic distributed hash table. As described above, contig generation is really a simple application split into two phases, an insert phase, which builds the hash table, and a traversal phase, which uses the hash table to traverse the de Bruijn graph of overlapping symbols. Because of this phasal behavior, we can optimize the performance of the hash table using BCL's hash map buffer, which aggregates hash table inserts with bulk insertions to local queues on the appropriate node, then transfers them to the hash table using a fast local operation once a flush operation is invoked. Our implementation of the Meraculous benchmark is only 600 lines long, 400 of which are for reading, parsing, and manipulating k-mer objects.

We implemented contig generation using the Meraculous algorithm [14, 15]. Our implementation is similar to the high-performance UPC implementation [15], but (1) uses our generic hash table instead of a highly specialized hash table and (2) uses a less sophisticated locking scheme, so sometimes processes may redundantly perform extra work by reconstructing an already constructed contig.

We benchmarked our hash table across the same three HPC systems described in Table 5 using the *chr14* (human chromosome 14) dataset. We compared our implementation to the high-performance UPC reference Meraculous benchmark implementation provided on the NERSC website, which we compiled with Berkeley UPC with hardware atomics enabled [2, 15]. We should note that the Meraculous UPC benchmark is based on the HipMer application, which may have higher performance [14]. We also compared our hash table to PapyrusKV, a high-performance general-purpose hash table implemented in MPI which has a Meraculous benchmark implementation available [21]. All performance results were obtained by with one process per core. Benchmarks for the UPC implementation are not available on Summitdev because the code fails on POWER processors due to an endianness issue. As shown in Figure 5, the BCL implementation matches or exceeds the performance of both the reference high-performance implementation of Meraculous and the general PapyrusKV hash table.

8.3 Microbenchmarks

We prepared a collection of microbenchmarks to compare (1) different backends' performance across data structure operations and (2) the relative performance of different implementations of data structure operations. Each benchmark tests a single data structure operation. Turning first to the `HashMap` microbenchmarks in Figure 6: we see clear differences between fully atomic versions of data structure operations (`find_atomic` and `insert`) and those with fewer atomicity guarantees or buffering (`find` and `insert_buffer`). We see that buffering offers an order of magnitude increase in performance, which is expected when transforming a latency-bound operation into a bandwidth-bound operation. The optimized `find` operation increases performance by 2-3x, as we would expect from the relative best-case costs ($2A + R$ and R).

The queue performance features two kinds of benchmarks: benchmarks looking at operations by all processes on a single queue, and benchmarks looking at operations by all processes on a collection of queues, one on each processor (the latter benchmarks are labeled "many"). In the `CircularQueue` benchmarks, we see that fully atomic operations (`pop_pushpop` and `push_pushpop`) are quite expensive when all processes are inserting into a single queue, compared to the less-atomic `pop_pop` and `push_push`. This is unsurprising, since the final CAS operation in an atomic `CircularQueue` push or pop essentially serializes the completion of operations. When pushing to multiple queues, using the less-atomic operation gives a factor of two performance improvement (`push_many_pushpop` vs. `push_many_push`).

In `FastQueue` benchmarks, we see that this data structure achieves significant performance improvements, even over the less-atomic implementations of `CircularQueue`'s methods.

Across all benchmarks, it appears that GASNet-EX is most effective at automatically utilizing local atomics when only running on

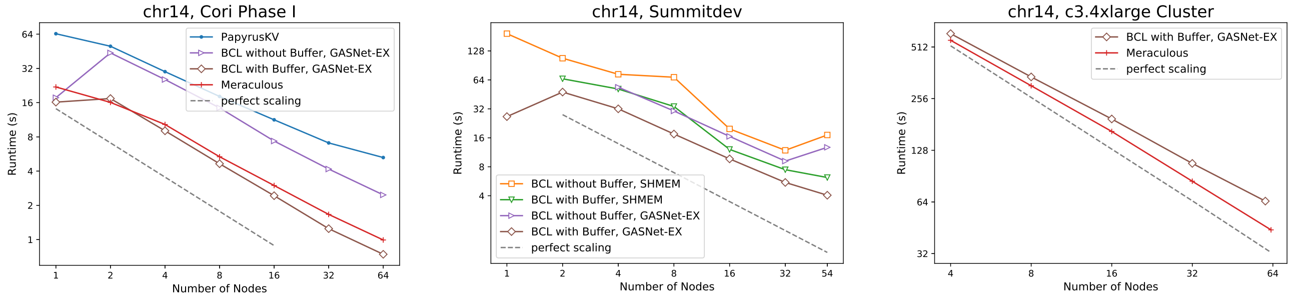

Figure 5: The Meraculous benchmark on the *chr14* dataset.

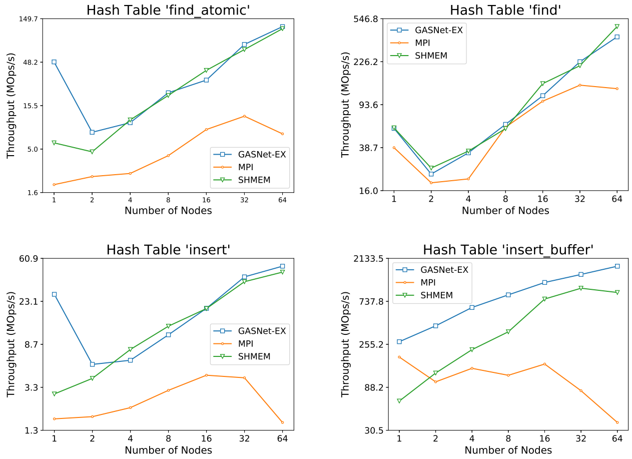

Figure 6: Hash table microbenchmarks.

one node, while MPI lags behind on most benchmarks, particularly those which make heavy use of atomic operations.

9 RELATED WORK

UPC, Titanium, X10, and Chapel are parallel programming languages which offer a PGAS abstraction [9, 11, 12, 29, 30].

UPC++ is a C++ library which offers a PGAS programming model [4]. UPC++ has a heavy focus on asynchronous programming that is absent from BCL, including futures, promises, and callbacks. UPC++'s remote procedure calls can be used to create more expressive atomic operations, since all RPCs are executed atomically in UPC++. However, these operations require interrupting the remote CPU, and thus have slower throughput than true RDMA atomic memory operations. The current version of UPC++ lacks a library of data structures, and UPC++ is closely tied to the GASNet communication library, instead of supporting multiple backends.

DASH is another C++ library that offers a PGAS programming model [13]. DASH has a large focus on structured grid computation, with excellent support for distributed arrays and matrices and an emphasis on providing fast access to local portions of the distributed array. While DASH's data structures are generic, they do not support objects with complex types. DASH is tied to the DART communication library, which could potentially offer performance portability through multiple backends, but is currently limited to MPI for distributed memory programs.

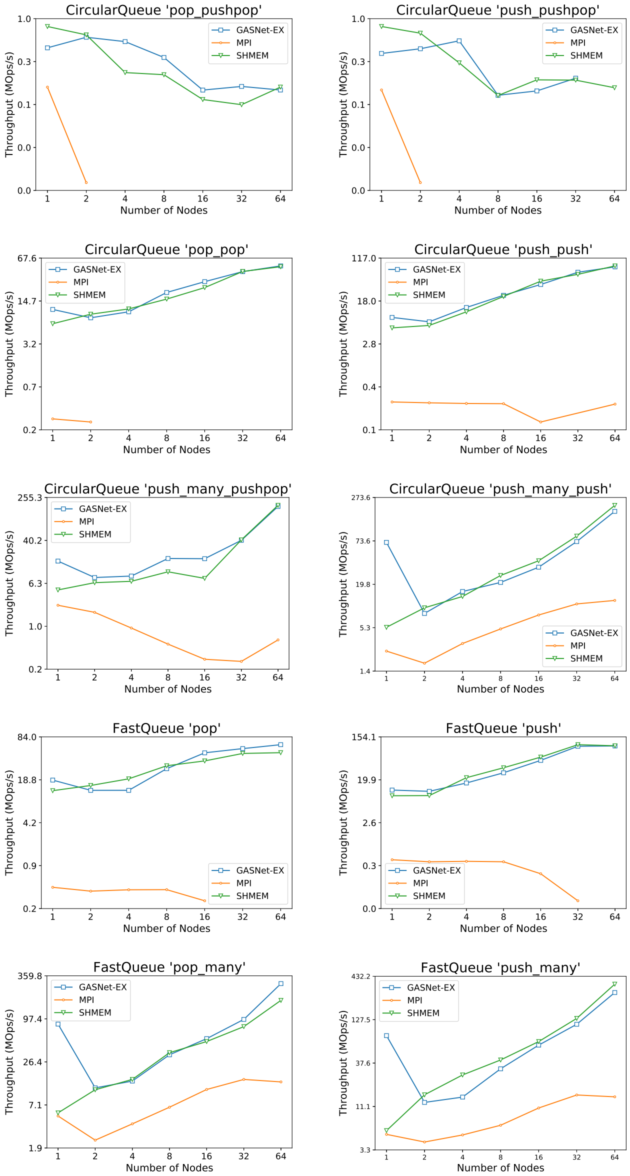

Figure 7: CircularQueue and FastQueue microbenchmarks.

HPX is a task-based runtime system for parallel C++ programs [20]. It aims to offer a runtime system for executing standard C++ algorithms efficiently on parallel systems, including clusters of computers. Unlike BCL, which is designed to use coordination-free RDMA communication, HPX's fundamental primitives are remote procedure calls used to distribute tasks.

STAPL, or the standard adaptive template library, is an STL-like library of parallel algorithms and data structures for C++ [28]. STAPL programs are written at a much higher level of abstraction than BCL, in a functional style using special higher-order functions such as map, reduce, and for-each which take lambda functions as arguments. From this program description, STAPL generates a hybrid OpenMP and MPI program at compile time. Some versions of STAPL also include a runtime which provides load balancing. The current version of STAPL is only available in a closed beta and only includes array and vector data structures [3].

The Multipol library provided a set of concurrent data structures on top of active messages, including dynamic load balancing and optimistic task schedulers [8]. However, it was non-portable and did not have the rich set of hash table data structures discussed here nor the notion of concurrency promises.

Global Arrays provides a portable shared memory interface, exposing globally visible array objects that can be read from and written to by each process [24]. While many application-specific libraries have been built on top of Global Arrays, it lacks the kind of high-level generic data structures that are the focus of this work.

10 CONCLUSION

BCL is a distributed data structures library that offers productivity through high-level, flexible interfaces but maintains high performance by introducing minimal overhead, offering high-level abstractions that can be directly compiled down to a small number of one-sided remote memory operations. We have demonstrated that BCL matches or exceeds the performance of both hand-optimized domain-specific implementations and general libraries on a range of benchmarks and is portable to multiple HPC systems.

ACKNOWLEDGMENTS

This work was supported by the NSF Graduate Research Fellowship under Grant No. DGE 1752814, by the NSF under Award No. 1823034, and by the DOE Office of Science under contract No. DE-AC02-05CH11231. It used resources of the National Energy Research Scientific Computing Center and the Oak Ridge Leadership Computing Facility, which are supported by the DOE Office of Science under Contract Nos. DE-AC02-05CH11231 and DE-AC05-00OR22725, respectively, as well as the AWS Cloud Credits for Research program. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these sponsors.

REFERENCES

[1] [n. d.]. Chapel ISx Benchmark. Retrieved March 10, 2018 from https://github.com/chapel-lang/chapel/tree/master/test/release/examples/benchmarks/isx

[2] [n. d.]. Meraculous Benchmark. Retrieved March 2, 2018 from http://www.nersc.gov/research-and-development/apex/apex-benchmarks/meraculous/

[3] 2017. STAPL Beta Release Tutorial Guide. (2017).

[4] John Bachan, Dan Bonachea, Paul H Hargrove, Steve Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, and Scott B Baden. 2017. The UPC++ PGAS library for Exascale Computing. In *Proceedings of the Second Annual PGAS Applications Workshop*. ACM, 7.

[5] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[6] Dan Bonachea and P Hargrove. 2017. GASNet Specification, v1. 8.1. (2017).

[7] B. Brock, A. Buluç, and K. Yelick. 2018. BCL: A Cross-Platform Distributed Container Library. *arXiv e-prints* (Oct. 2018). arXiv:cs.DC/1810.13029

[8] Soumen Chakrabarti, Etienne Deprit, Eun-Jin Im, Jeff Jones, Arvind Krishnamurthy, Chih-Po Wen, and Katherine Yelick. 1995. Multipol: A distributed data structure library. In *PPoPP*.

[9] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[10] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Conf. on Partitioned Global Address Space Programming Models*. ACM, 2.

[11] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, Vol. 40. ACM, 519–538.

[12] UPC Consortium et al. 2005. UPC language specifications v1. 2. *Lawrence Berkeley National Laboratory* (2005).

[13] Karl Fürlinger, Tobias Fuchs, and Roger Kowalewski. 2016. DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms. In *HPCC*. Sydney, Australia, 983–990. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0140

[14] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2015. HipMer: an extreme-scale de novo genome assembler. In *Int. Conf. for High Perf. Comp., Networking, Storage & Analysis (SC)*. ACM, 14.

[15] Evangelos Georganas, Aydin Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2014. Parallel de bruijn graph construction and traversal for de novo genome assembly. In *Int. Conf. for High Perf. Comp., Networking, Storage & Analysis (SC)*. IEEE Press, 437–448.

[16] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2014. Enabling highly-scalable remote memory access programming with MPI-3 one sided. *Scientific Programming* 22, 2 (2014), 75–91.

[17] Ulf Hanebutte and Jacob Hemstad. 2015. ISx: A scalable integer sort for co-design in the exascale era. In *Conf. on Partitioned Global Address Space Programming Models*. IEEE, 102–104.

[18] Jacob Hemstad, Ulf R Hanebutte, Ben Harshbarger, and Bradford L Chamberlain. 2016. A study of the bucket-exchange pattern in the PGAS model using the ISx integer sort mini-application. In *PGAS Applications Workshop (PAW) at SC16*.

[19] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.

[20] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A task based programming model in a global address space. In *Conference on Partitioned Global Address Space Programming Models (PGAS)*. ACM, 6.

[21] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter. 2017. PapyrusKV: a high-performance parallel key-value store for distributed NVM architectures. In *Int. Conf. for High Perf. Comp., Networking, Storage & Analysis (SC)*. ACM, 57.

[22] Timothy G Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for parallel programming*. Pearson Education.

[23] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured parallel programming: patterns for efficient computation*. Elsevier.

[24] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. 1996. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing* 10, 2 (1996), 169–189.

[25] Robert W Numrich and John Reid. 1998. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, Vol. 17. ACM, 1–31.

[26] Chao-Wei Ou, Sanjay Ranka, and Geoffrey Fox. 1996. Fast and parallel mapping algorithms for irregular problems. *The Journal of Supercomputing* 10, 2 (1996), 119–140.

[27] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient Bloom filters. *Journal of Experimental Algorithmics (JEA)* 14 (2009), 4.

[28] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. 2011. The STAPL Parallel Container Framework. In *PPoPP*. ACM, 235–246.

[29] Michele Weiland. 2007. Chapel, Fortress and X10: novel languages for HPC. *EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706* (2007).

[30] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phillip Colella, et al. 1998. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience* 10, 11-13 (1998), 825–836.