DArray: A High Performance RDMA-Based Distributed Array

Baorong Ding
dingbaorong@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Mingcong Han
mingconghan@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai AI Laboratory
Shanghai, China

Rong Chen
rongchen@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai AI Laboratory
Shanghai, China

ABSTRACT

This paper presents DArray, a high performance RDMA-based distributed memory system. DArray achieves high performance through three key designs. First, DArray is designed with an object array abstraction, which captures the high-level application semantics and provides a rich set of optimized interfaces with object granularity. Second, DArray adopts distributed cache to absorb remote data accesses. In order to reduce the performance overhead incurred by the cache layer and increase the parallelism, DArray devises a lock-free data access path to the local cache which utilizes reference counters to prevent data races. Finally, based on the observation that most data update operators are associative and commutative, DArray proposes a new "Operate" interface, which enables concurrent data operations on multiple nodes, and extends existing distributed cache coherence protocol to support the new "Operate" semantics.

A graph analytics engine and a distributed key-value store (KVS) are built on top of DArray to demonstrate its versatility. The experiment results on graph applications show that DArray achieves two to three orders of magnitude better performance than GAM (a state-of-the-art distributed memory), also with a maximum speedup of 2.1x compared to Gemini (a state-of-the-art distributed graph engine). Furthermore, the DArray-based KVS outperforms the GAM-based KVS by up to 41x (from 2x).

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**.

KEYWORDS

RDMA, Distributed Shared Memory, Distributed Data Structure, Distributed System

ACM Reference Format:
Baorong Ding, Mingcong Han, and Rong Chen. 2023. DArray: A High Performance RDMA-Based Distributed Array. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605608

1 INTRODUCTION

Distributed in-memory processing is becoming increasingly popular, as it enables computation on large datasets to be carried out more efficiently. Distributed shared memory (DSM) systems are widely used for building distributed in-memory processing applications, as they provide a unified view of memory in distributed machines. The low-level byte-addressed memory model of DSM makes it more flexible for applications that require fine-grained data access (e.g., graph analytics and distributed key-value stores), compared to bulk-based data processing systems like Dryad[15], Spark [21]. Programming distributed applications on DSM is as easy as on a multicore system. Therefore, many distributed shared memory systems have been developed to support distributed applications, such as Argo [17] and Magi [14], especially with the emergence of low-latency RDMA networks.

Despite the flexibility in programming made possible through low-level abstraction in distributed shared memory systems, this comes with limitations that hinder the DSM's ability to capture high-level application semantics. For example, when locks are used to avoid data race, DSM is not certain which objects the lock is protecting. As a result, traditional DSM systems often perform pessimistic synchronization of all pages to prevent possible conflicts, negatively affecting DSM system performance. To mitigate this performance issue, providing high-level application semantic can be helpful. Through high-level semantics, more precise data can be obtained about the objects protected by the lock.

To this end, this paper presents DArray, a high performance RDMA-based distributed memory system that takes advantage of the high-level application semantics. Instead of byte-addressed read/write, DArray utilizes an abstraction of a global object array spanning multiple nodes. With the object array abstraction, DArray provides a rich set of interfaces with object granularity, such as get/set and R/W lock. To improve performance, DArray adopts various optimization techniques.

First, DArray is designed with a cache layer to exploit data locality. Each node maintains a local cache to store the most recently accessed objects, which is managed by a runtime system with cache coherence protocol. In order to minimize the performance overhead incurred by the cache layer on the critical path, DArray has an elaborately designed data access path. Unlike lock-based solutions presented in existing cache-based systems [5] to prevent data races between application threads and runtime threads, DArray employs a lock-free data access path that enables application threads to access cached data just holding a reference (implemented with atomic variables). To improve performance even further for appropriate sequential access scenarios, DArray also provides an optional "Pin" interface that holds the reference of a chunk explicitly for subsequent data access, reducing the number of atomic variable read/write significantly and achieving data access performance comparable to native arrays.

Second, DArray proposes the "Operate" interface to better cater to application demands, which performs an atomic read-then-write operation on an object. With the observation that many operators (e.g., `write_add`, `write_min`) are both associative and commutative, DArray leverages an optimization that locally combines the operations on the same object, and moves the combined operation to the home node of the object, which can significantly improve the parallelism. However, this new "Operate" interface doesn't fit well in existing cache coherence protocols. To facilitate such optimizations, we extend the existing cache coherence protocol with a new "Operated" state, which enables concurrent data operations on multiple nodes, in contrast to the exclusive ownership restriction imposed by the "Write" interface.

We have implemented DArray as a header-only user-space library, comprising about 5,000 lines of C++ code. Several micro benchmark results demonstrate that DArray outperforms GAM [5] (a state-of-the-art distributed memory system) and BCL [4] (a distributed data structure library) in both performance and scalability. We also demonstrate the versatility of DArray as a distributed data structure by implementing two distributed applications: a graph analytics engine and a distributed key-value store (KVS) with the interfaces provided by DArray. Benchmarks conducted on graph applications show that DArray achieves two to three orders of magnitude better performance than GAM, also with a maximum speedup of 2.1x compared to Gemini [23] (a specialized distributed graph analytics engine). Furthermore, the DArray-based KVS outperforms the GAM-based KVS by 2x to 41x.

The contributions made in this paper are summarized below:

(1) The design of DArray, a high performance distributed object array with a rich set of high-level interfaces.
(2) The design of a cache layer with a lock-free data access path that achieves low overhead and high degree of parallelism in the presence of local cache and a runtime system.
(3) The design of a new "Operate" interface that enables concurrent data operations on multiple nodes along with an extended distributed cache coherence protocol.
(4) Two distributed applications built on the interfaces provided by DArray to demonstrate its versatility and a set of evaluations that confirm the efficacy of DArray for distributed data-intensive applications.

2 MOTIVATION AND DESIGN PRINCIPLE

Implementing a high performance distributed memory system is challenging. We summarize three key principles that motivate the design of DArray as follows.

Take advantage of locality. As is well-known, exploiting locality can effectively improve system performance [9]. One mechanism that utilizes this principle of locality is the cache. To demonstrate the significance of cache for distributed memory, we first evaluate the performance of two different RDMA-based distributed memory systems, GAM (a distributed memory system with cache), and BCL (a distributed data structure without cache), through sequentially accessing an array. As shown in Figure 1, in a distributed workload with 6 nodes, BCL exhibits significantly higher data access latency compared to GAM. This is primarily because BCL lacks local cache, necessitating network communication for each access and resulting in a latency that is comparable to that of RDMA one-sided READ (2μs). While GAM incorporates a local cache for remote access scenarios, reducing the frequency of network communication and resulting in a lower latency than that of BCL. Therefore, to achieve better performance, DArray also adopts a cache mechanism.

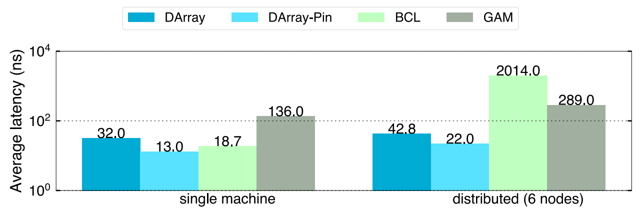

Figure 1: Average latency of 8-byte sequential access over the entire array. In the distributed scenario, the array is evenly distributed among these nodes.

However, though GAM has lower remote access latency than BCL, it has higher latency when accessing local memory, as shown in the single machine configuration of Figure 1. This is because introducing a cache layer increases the path to access local data, leading to significant performance overhead. Thus, implementing a cache layer with low overhead is still challenging.

Capture application semantic. Another principle is to use the semantic information of data access from applications to optimize the performance of the memory system. For example, knowing the object access pattern can help the cache layer avoid repeatedly migrating the data between nodes, thus reducing the overhead of cache maintenance and improving performance. As shown in Figure 1, DArray-Pin is implemented with a memory access pattern hint using the pin interface (described in §4.1), so it has a lower latency than DArray. Therefore, in order to capture the application semantics of data accesses, DArray provides various interfaces to achieve extreme performance.

Utilize RDMA networks. The high-throughput and low-latency features of RDMA networks are essential for developing a high performance distributed memory system. However, these benefits don't come for free. RDMA has a different programming model than traditional TCP/IP networks, necessitating efficient mapping of various network communications (e.g., application data, coherence requests) in DArray to the primitives provided by RDMA. Furthermore, several optimization techniques for RDMA networks are also crucial to the high performance of DArray.

3 DARRAY OVERVIEW

3.1 Architecture

Figure 2 presents an overview of DArray's architecture. DArray provides APIs for distributed applications to be built on and an abstraction of a global array spanning multiple nodes. These nodes are connected using low-latency RDMA networks. DArray adopts a layered design, with each layer having its specific responsibility. These layers communicate with each other through lock-free

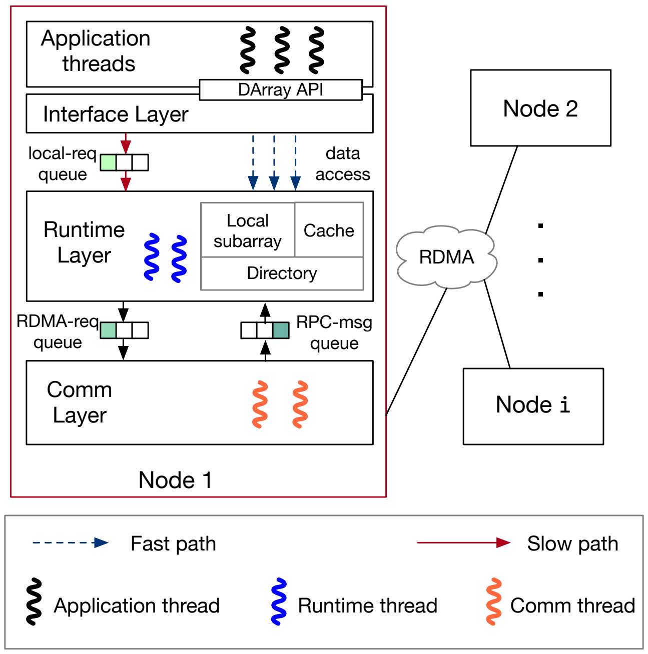

Figure 2: Overview of DArray architecture. All application threads of these nodes are in the same distributed application (e.g., graph analytics, key-value store).

queues and specific interfaces. Starting from the top, these layers are interface layer, runtime layer, and communication layer.

Interface layer. The interface layer is designed to provide low-overhead abstraction and maximum concurrency for data access that is directly called by application threads. It provides APIs of rich semantics to applications, and utilizes the functionalities provided by the runtime layer. To fulfill applications' needs, it attempts to access the local subarray or cache directly first. If the data is not available, it submits a request to the runtime layer via local-request queue and awaits the request to be fulfilled.

Runtime layer. The runtime layer takes the responsibility of handling requests from both local and remote nodes, cache management, executing state transitions conforming to the extended cache coherence protocol, and implementing various runtime optimizations. The local subarray is the application data allocated by the local node, while the cache stores local copies of remote data. The directory [1] tracks the state of data in both local subarray and cache at the chunk granularity (512 elements by default) and is used by the cache coherence protocol to maintain data consistency. The runtime layer receives requests through the local-request queue and RPC-message queue, and sends RDMA requests to the communication layer via the RDMA-request queue. By separating request handling and network communication, it allows for the overlap of computation and communication and masks network latency that cannot be ignored.

Communication layer. The communication layer provides RDMA communication support for the runtime layer and consists of two types of networking threads. Tx (Transmit) threads are responsible for receiving RDMA requests from the runtime and posting them to the RNIC (RDMA NIC). While Rx (Receive) threads constantly poll the RNIC and deliver received RPC messages to the runtime via the RPC-message queue. Additionally, these networking threads perform RDMA-related optimizations, such as selective signaling and batching. The separation of Tx and Rx threads ensures efficient and high-throughput network communication.

[1]Directory in DArray has similar functionalities to that of a directory employed in directory-based CPU cache coherence protocol.

```
template <class T>
class DArray {
  using operator_func = void(T&, T);
▸ Constructor
  DArray(size_t n, vector<size_t> partition_offset={});
▸ Read/Write API
  T get(size_t index);
  void set(size_t index, const T& new_val);
▸ Concurrency control
  void RLock(size_t index);
  void WLock(size_t index);
  void UnLock(size_t index);
▸ Operate API
  int registerOp(operator_func operator);
  void apply(size_t index, int op_id, const U& operand);
▸ Optimization hint
  bool pin_data(size_t index, OpType op);
  bool unpin_data(size_t index);
};
```

Figure 3: DArray APIs.

3.2 Interface

DArray, as a data structure, provides a rich set of APIs, as shown in Figure 3. These APIs can be classified into five categories.

Constructor. The constructor of DArray creates a distributed array with n elements that spans multiple nodes. By default, the global array is evenly partitioned among these nodes. However, users have the option to specify a custom partition scheme by providing the optional argument, `partition_offset`.

Read/Write API. Similar to existing distributed data structures, DArray provides basic Read/Write APIs that enable access to the global array.

Concurrency control. For parallel and distributed applications, locks play a crucial role in avoiding data races. Therefore, DArray also provides distributed reader/writer locks for these applications to ensure data consistency.

Operate API. The Operate API is a unique design of DArray that utilizes the associativity and commutativity found in many operators, and its detail will be discussed in §4.3. Applications can register custom operators with the DArray runtime via the `registerOp` method, and the runtime will assign operator IDs. Applications can use these IDs to invoke corresponding operators to operate data.

Optimization hint. To minimize abstraction overhead (overhead present in data access path), applications can provide optimization hints to the DArray runtime using the Pin interface, which will be further discussed in §4.1.

4 DESIGN AND OPTIMIZATION

4.1 Data Access

To achieve the objective of a high performance distributed memory system, DArray first needs to design a low-overhead data access path, as applications with good data locality are more sensitive to the abstraction overhead.

However, the incorporation of local cache complicates the data access path. This is because there may exist data races between the application thread that accesses data and the runtime thread that manages it. For instance, while an application thread is writing to its cached data, the runtime thread may be evicting it back to its home node[2], causing a loss of data updates. Similar situations can also happen for data in the local subarray.

Lock-based approach. One strawman solution is to introduce locks in both data access path and runtime management to prevent possible data races. This approach is simple but has several drawbacks:

- **Large overhead.** It introduces locks in each data access, which is unacceptable for many applications with good locality, since the network latency is already amortized.
- **Limited concurrency.** The concurrency of application threads is restricted, as only one application thread is allowed to access data in the same chunk at a time. The possible presence of false sharing [16] in applications may exacerbate this issue of limited concurrency.
- **Restriction on optimizations.** Some runtime management operations, such as updating the chunk's state from readable to writable, do not conflict with existing data access from application threads and can coexist with them. However, in the lock-based approach, these parallelisms are suppressed.

Therefore, DArray takes a different approach where the interaction between these threads is carefully designed to minimize overhead on the critical path. For the data access path of application threads, lock-free reference counting is adopted to reduce overhead and maximize parallelism. In contrast, lock-based concurrency control is used in the runtime thread, which is not sensitive to performance, to simplify the design. Figure 4 and Figure 5 respectively demonstrate the data access path in application threads and the cache eviction path in runtime threads.

Data access path. When an application thread invokes the `get` API, it determines whether to read from the local subarray or the cache, based on whether the data is allocated on the local node (Figure 4, lines 1–4). To simplify the discussion, we only show the path for reading from the cache. In this case, the `index` is used to locate the directory entry (`dentry`) that manages the chunk where the `index` belongs (line 5). The `delay_flag` is checked (line 6) to prevent runtime threads from starving, and if it is set, the thread will spin and wait (line 7). Afterward, the reference counter is increased to prevent state transitions between time-of-check and time-of-use (line 8). If the state of the dentry is readable, it remains so (line 13) and it's safe to read from this chunk until the reference counter is decreased (line 14). However, if the state of the dentry is not

```
▸ Read interface
get(size_t index) -> T
1   if isLocal(index)            ▸ allocated by local node?
2   | return read_in_local_subarray(index);
3   else
4   | return read_in_cache(index);

▸ Read in local cache
read_in_cache(size_t index) -> T
5   auto& dentry = get_dentry(index);
6   while dentry.get_delay_flag()
7   | spin();               ▸ prevent runtime from starving
8   dentry.inc_refcnt();              ▸ hold a reference
9   while !dentry.readable()
10  | dentry.dec_refcnt();
11  | dentry.wait_for_cache_fill();
12  | dentry.inc_refcnt();
13  auto ret = dentry.get(index);          ▸ read the data
14  dentry.dec_refcnt();           ▸ release the reference
15  return ret;
```

Figure 4: Data access in DArray.

```
▸ Cache eviction in runtime
evict_entry(Dentry& dentry) -> void
▸ prevent data races in runtime
1   std::lock_guard lk(dentry);
2   dentry.set_delay_flag();  ▸ block incoming app threads
3   dentry.set_state(INVALID);          ▸ state transition
4   dentry.wait();            ▸ wait for refcnt to reach 0
▸ unblock incoming app threads
5   dentry.clear_delay_flag();
▸ safely evict cacheline
6   evict_cacheline(dentry.cacheline);
```

Figure 5: Runtime management in DArray, take cache eviction as an example.

```
▸ Got a promotion from read-only to writable
granted_write_permission(Dentry& dentry) -> void
1   std::lock_guard lk(dentry);
▸ safely change its state without syncing
2   dentry.set_state(DIRTY);
```

Figure 6: Runtime management in DArray, take permission promotion as an example.

readable, the application thread will send a request to the runtime and wait for it to be fulfilled (line 11).

Runtime management path. The runtime management path is designed to be much simpler since it is not on the critical path. Let's take cache eviction as an example. The lock is acquired to prevent data races between runtime threads (Figure 5, line 1). When the dentry's state needs to be changed, the following four steps are performed: ① The delay flag is set to put all upcoming application threads accessing this chunk on hold (line 2). ② The dentry's state is changed to the new state, which in this example is `INVALID` (line 3). ③ The runtime thread waits for all existing application threads that have acquired references to this chunk to finish their data accesses (line 4). ④ Finally, the delay flag is cleared to unblock incoming application threads (line 5). Now, it's safe to truly evict a cacheline (line 6).

Optimization of parallelism. In certain scenarios, accessing a chunk's data and modifying its state can be parallelized. For instance, as demonstrated in Figure 6, when the permission of a dentry is promoted from read-only to writable, existing data access is not

[2]Home node represents the node where this chunk is allocated.

affected. Consequently, the runtime thread can directly modify the state of the chunk without synchronization with user threads (line 2).

The proposed data access design reduces abstraction overhead, increases parallelism, and adds room for optimizations.

- **Minimal overhead.** In the best case, the overhead introduced by DArray compared to builtin arrays is only a single atomic variable read (`delay_flag`), two atomic variable writes (`refcnt`), and some branch instructions. Moreover, in certain scenarios, the overhead of atomic variables can be completely eliminated, which will be discussed later in this section.
- **Improved concurrency.** The lock-free data access permits multiple application threads to access the same chunk concurrently, resulting in improved concurrency compared to the lock-based approach.
- **Room for optimizations.** In certain scenarios (e.g., permission promotion), application threads and runtime threads referring to the same chunk can be parallelized.

Pin interface. Despite a variety of optimizations made to minimize the abstraction overhead of DArray's data access path, there is still some overhead from atomic variables in the fast path [3]. Atomic variables are used to prevent the runtime altering the state of a chunk while application threads are accessing it. In scenarios with sequential access, a chunk is often accessed repeatedly, and the chunk's state is unlikely to be changed during this period. To address this, we offer the "`pin`" API that ensures the chunk's state remains unchanged until the "`unpin`" API is invoked. Under the hood, pinning a chunk means holding its reference (`refcnt` remains nonzero), so runtime cannot evict it (for cached data) or degrade its permission (e.g., from writable to only-readable). However, if it is pinned to be read (with "Shared" state in §4.4), runtime can share its data to fulfill other nodes' read requests. Consequently, the use of the "`pin`" API eliminates the need for atomic variables in the fast path of "`get`", "`set`", and "`apply`", since "`pin`" has explicitly held the reference for them. To use "`pin`", applications should have some knowledge about their access pattern.

4.2 Cache Management

Cache is a critical component in the design of DArray. It can effectively reduce network communication caused by remote data access, particularly when applications exhibit good data locality. Cache management is handled by the runtime, including maintenance of cache coherence, cache eviction, and prefetching.

Cache coherence. In order to maintain cache coherence, runtime threads receive requests from both local and remote nodes and update dentry's state accordingly. Dentry state transitions are defined in the extended cache coherence protocol, which will be further discussed in §4.4.

Cache eviction. To keep memory usage at a reasonable level and achieve low response time for obtaining free cachelines, a cache eviction mechanism to invalidate cold cachelines is required. Although LRU (Least recently used) is a commonly used cache replacement algorithm, its update involves complex hash table and

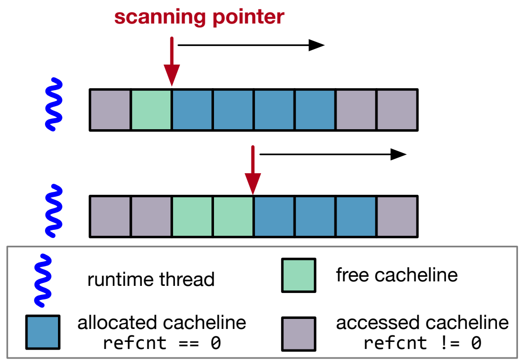

Figure 7: Cache eviction mechanism. Allocated cacheline is the cacheline with valid data but no application thread is accessing it, which can be evicted. Accessed cacheline is the cacheline with valid data and some application threads are accessing it, which cannot be evicted.

linked list operations, which can introduce unacceptable overhead and limit concurrency in the data access path. In DArray, we avoid introducing additional overhead in the data access path and delegate cache eviction entirely to the runtime. As shown in Figure 7, each runtime thread has its own independent cache region and a corresponding scanning pointer, which allows DArray to avoid data races and increase concurrency. The cache eviction policy is governed by two parameters: *low watermark* and *high watermark*. When the number of free cachelines in a local cache is below *low watermark*, the runtime thread will perform cache reclamation until the number of free cachelines is above *high watermark*. The default ratio of *low watermark* and *high watermark* is 30% and 50%, which can be adjusted in configurations. During the reclamation process, the runtime thread scans its own cache region using the scanning pointer. When a scanned cacheline is not in an intermediate state[4] and has a refcnt of 0, it will be evicted.

Cache prefetch. As is widely recognized, sequential access is much faster than random access, which is why many applications are optimized for it. To better integrate with existing application optimization techniques, we introduce a prefetch mechanism in the slow path of DArray (Figure 4, line 17). The prefetch mechanism is placed in the slow path of data access for two reasons. Firstly, placing it in the fast path would significantly reduce data access performance and contradict our goal of minimal abstraction overhead. Secondly, when waiting for the cacheline to be filled in the slow path, the application thread is unable to make progress. Thus, the insertion of the prefetch mechanism in the slow path does not add an additional burden. The number of cachelines that need to be prefetched can be configured.

4.3 Operate Semantics

While the basic `Read`/`Write` interfaces are useful in general, many applications require more expressive interfaces. Inspired by widely-used combiner in distributed systems implemented by message

[3]As shown in Figure 2, fast path does not require waiting for runtime to handle the request, while slow path does.

[4]The intermediate state, which means it is still waiting for other node's reply, is introduced in implementing cache coherence protocol

```
PageRank(...)
▸ Register custom operator
1   auto inc_op_func = [](double& val, double inc)
                          { val += inc; };
2   int inc_op_id = DArray::registerOp(inc_op_func);
▸ allocate global array of vertex data
3   DArray<double> curr_rank(G.n);
4   DArray<double> next_rank(G.n);
▸ initialize vertex data
5   curr_rank.fill(1.0 / G.n);
6   next_rank.fill(0);
▸ core algorithm
7   for iter in [0:10]
8   | par_for src in [0:G.n]
9   | | par_for dst in G.V[src].neighours
10  | | | double inc = curr_rank.get(src) /
    | | |              G.V[src].neighours.size();
    | | | ▸ propagate rank value to neighbors
11  | | | next_rank.apply(dst, inc_op_id, inc);
    | ▸ prepare for the next iteration
12  | swap(curr_rank, next_rank);
13  | next_rank.fill(0);
```

Figure 8: An example of PageRank application using the Operate interface.

passing [7, 8, 19], we propose a new interface with "Operate" semantics, which better facilitates various data operations, including `write_add` and `write_min`.

$$val \oplus arg1 \oplus arg2 = val \oplus (arg1 \oplus arg2) \qquad (1)$$

As depicted in Equation 1, the fundamental concept of "Operate" semantics will be explicated by using the `write_add` operation as an illustration. To increment the value of `val` by `arg1` and `arg2`, two approaches can be applied: (a) apply the computation on the left side, where `val` is first incremented by `arg1` and then by `arg2`, or (b) apply the computation on the right side, where the sum of `arg1` and `arg2` is first computed and then added to `val`. The order of computation does not affect the final result.

The "Operate" interface is compatible with any operator that satisfies the properties of associativity and commutativity. Applications are required to register the operator to the DArray runtime and obtain an operator ID, which only needs to be passed along with the operand when calling the `apply` API.

We explain the "Operate" interface using the PageRank algorithm [3] as a case study, as depicted in Figure 8. PageRank is an algorithm that computes the rank of each vertex based on the ranks of its neighbors. In each iteration of PageRank, vertices evenly distribute their rank among their outgoing neighbors, which results in an increment of the neighbor vertices' ranks by `inc`. Initially, the PageRank application registers the `inc_op_func` to the DArray runtime and obtains an operator ID (`inc_op_id`) assigned by the runtime (line 2). During the iteration process, using the `apply` API, the `write_add` operation can be carried out by feeding both `inc_op_id` and `inc` as input (line 11).

Without the Operate API, it would be necessary to acquire the writer lock for the corresponding vertex, read the vertex's rank, add the increment value to the rank, and write it back before releasing the lock. The Operate interface offers some benefits over this approach. The writer lock is exclusive, which means that only one node can hold it at a time, leading to higher contention and limiting concurrency. In contrast, the Operate interface is not exclusive, as will be discussed in detail in §4.4, which can reduce contention and increase concurrency.

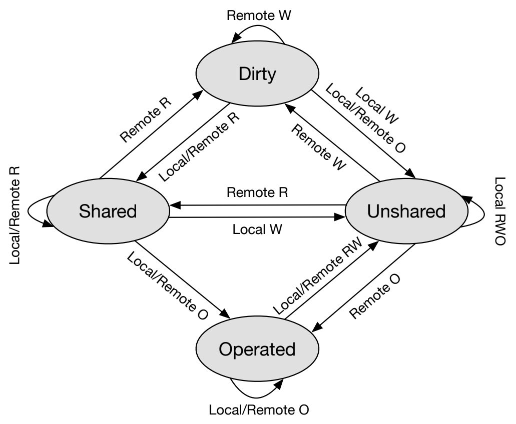

Figure 9: State machine of extended cache coherence protocol. Local/Remote is relative to the home node. RWO represents Read/Write/Operate.

Table 1: States in the cache coherence protocol. RWO represents Read/Write/Operate.

States	Home node	Other nodes	Exclusive
Unshared	R/W/O	None	Yes
Shared	R	R	No
Dirty	None	R/W or None	Yes
Operated	O	O	No

4.4 Extended Cache Coherence Protocol

Cache coherence protocol is essential in offering a consistent view of data to applications. While existing cache coherence protocols mainly focus on the Read/Write interface, it is not adequate to implement our proposed Operate interface. Consequently, we present an extended cache coherence protocol that supports the Operate interface. The extended cache coherence protocol comprises four states and their corresponding transitions, as shown in Figure 9.

Similar to existing protocols, our protocol is directory-based and ensures sequential consistency. This is because we don't buffer or reorder reads/writes, and all "Operate" operations are visible for subsequent reads/writes with happen-before relationships.

Table 1 presents a summary of the four states in the cache coherence protocol:

- **Unshared**: The ownership of the chunk is exclusively assigned to the home node. Subsequently, the home node has the permission to `Read/Write/Operate` this chunk's data.
- **Shared**: The chunk is shared among nodes, allowing all of them to `Read` the data.
- **Dirty**: A non-home node has exclusive ownership of the chunk, enabling it to `Read/Write` this chunk's data.
- **Operated**: The chunk is shared among all nodes to enable concurrent `Operate` operations on the data. These operations will be merged later by the home node (the node where this chunk is allocated).

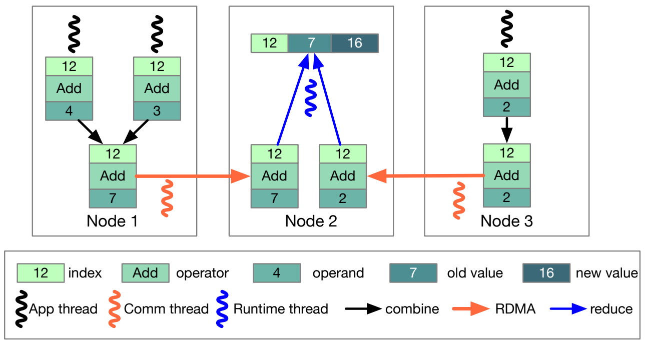

Figure 10: Logical process of Operate interface.

Given that the other three states resemble existing cache coherence protocols and are easily comprehensible, this section focuses primarily on explaining the proposed "Operated" state. Figure 10 shows the logical process of the Operate interface. In the Operated state, each node is capable of applying a registered operator to the data in this chunk. Since operators are associative and commutative, their operands are first combined in the local cache. Once the cacheline is evicted, the combined operands are written back to the home node, which then applies the received operand to local data. Due to the potential incompatibility between different operators, the Operated state is uniquely associated with specific operator IDs to ensure correctness of the Operate semantics.

Also, the newly proposed "Operated" state can be well-coordinated with existing states.

Transitions to "Operated". A chunk in the "Shared" or "Unshared" state can easily transition to the "Operated" state. A chunk in the "Dirty" state has to write its dirty data back to its home node before transitioning to the "Operated" state, since it is the home node that has to perform the reduction based on other nodes' operations.

Transitions from "Operated". A chunk in the "Operated" state cannot be accessed, since no node has complete information to deduce its current value. If a read/write request is encountered, it transitions to the "Unshared" state, allowing the home node to gather data modifications (see Figure 9, transitions between "Operated" and "Unshared"). What follows is similar to access an "Unshared" chunk.

4.5 RDMA-Based Acceleration

Due to its low latency and zero-copy capabilities, we choose to implement DArray using RDMA networks. Our DArray implementation handles two forms of data transfer between nodes: application data and protocol messages. Specifically, we make use of one-sided RDMA `WRITE` *verb* to transmit application data and two-sided RDMA `SEND/RECV` *verbs* to transmit protocol messages. Moreover, we apply various optimizations related to RDMA.

Dedicated networking threads. Some networking threads are dedicated to interacting with RNIC (RDMA NIC), while other threads offload network communication via RDMA-request queue. This eliminates the need for connections between every pair of threads, reducing the required number of queue pairs (RDMA connection) from $n^2 \times t$ (where t refers to the number of threads in each node) to $n^2 \times c$ (where c refers to the number of networking threads in

```
1   DArray<Entry> hashtable(kEntries);
2   DArray<uint8_t> byte_array(kMemSz);
▸ Get a key_value pair from the key-value store
get(key_t key) -> key_val_t
3   key_val_t kv = {};              ▸ initialize it to empty
4   int bucket_id = hash(key);          ▸ hash key to bucket
5   int start = ...;            ▸ start entry in this bucket
6   int end = ...;
7   bool found = false;
8   for idx in [start:end]
9   │ hashtable.RLock(idx);
10  │ Entry entry = hashtable.get(idx);
11  │ if match_key(entry, key)
12  │ │ found = true;
13  │ │ kv = KeyVal(byte_array.get(entry.addr));
14  │ │ hashtable.Unlock(idx);
15  │ │ break;
16  │ hashtable.Unlock(idx);
▸ try to find in extra buckets
17  if (!found)
18  │ size_t overflow_ptr = hashtable.get(end+1);
19  │ ...                        ▸ follow overflow pointer
20  return kv;
```

Figure 11: Simplified code for implementing distributed key-value store using DArray's API.

each node). This optimization can help avoid RNIC's on-chip cache misses, as on-chip cache sizes are often limited.

Selective signaling. By default, RNIC generates a *work completion* upon completion of every *work request*. This requires networking threads to poll them, resulting in much PCIe traffic caused by MMIO Reads. We employ an optimization of selective signaling, which instructs RNIC to produce *work completions* only for every r requests, with the exception of some RDMA requests that require signals to reclaim buffers. The optimization technique helps reduce the PCIe traffic.

5 APPLICATIONS

In this section, we showcase how DArray's abstraction can be used to construct distributed applications through the development of two applications: a graph analytics engine and a distributed key-value store.

5.1 Graph Analytics

A single-machine graph analytics engine (e.g., Polymer [22]) builds on built-in shared-memory arrays for communication between different computational units. However, a distributed graph analytics engine (e.g., Gemini [23]) relies on explicit message passing, given that there is no built-in shared-memory abstraction among the nodes. To port a single-machine graph analytics engine to a distributed one, we could simply replace the built-in arrays with our DArray, which provides shared-memory abstraction among the nodes, and reuse the computation engine and task scheduling components of the graph analytics engine. Figure 8 is a simplified version of the PageRank algorithm based on DArray and omits other parts that require domain knowledge of graph processing.

5.2 Distributed Key-Value Store

The distributed key-value store is an important distributed application that can be conveniently constructed on top of the DArray abstraction. A distributed key-value store comprises an entry array and a byte array, both spanning multiple nodes (Figure 11, lines

1–2). The entry array is partitioned into buckets, with each bucket containing 15 entries and an overflow pointer, and the hash function maps a key to a particular bucket. Each entry is 8 bytes and comprises an 8-bit `tag`, 16-bit `size`, and 40-bit `offset`. The `tag` distinguishes entries within a bucket, while the `size` indicates the size of the key-value pair, and the `offset` represents the byte offset of the key-value pair within the byte array. The overflow pointer is used to chain extra bucket when this buckets is full. We port the SlabAllocator from Memcached [11] to manage the byte array.

Figure 11 demonstrates how to retrieve a key-value pair from this key-value store. ① First, the hash function maps the key to a particular bucket (line 4). ② Afterwards, we probe each entry in this bucket to find a match (lines 8–16). ③ If not found, we follow the `overflow_ptr` to probe an extra bucket until there exists no extra bucket (lines 17–19). ④ Finally, we return either the key-value pair we found or an empty one (line 20).

6 EVALUATION

6.1 Experimental Setup

Implementation. DArray is implemented as a header-only user-space library comprising approximately 5,000 lines of C++ code, which enables easy integration with various applications.

Testbed setup. The experiments are conducted on an RDMA-capable cluster with twelve nodes. Each node is equipped with two Intel Xeon E5-2650 v4 CPUs (total 2 × 12 cores), 128GB DRAM (128 GB/s, measured by Intel Memory Latency Checker [1]), and one ConnectX-4 100 Gbps InfiniBand RNIC. These servers' software environment is configured with GCC 11.2, OpenMPI 4.1.1, and Ubuntu 16.04. OpenMPI is configured with ucx to utilize high-speed RDMA networks.

Comparing targets. DArray-Pin is a variant of DArray that utilizes the Pin interface to improve performance. We also introduce BCL [4], GAM [5], and Gemini [23] for comparison. BCL is a distributed data structure library that maps access to remote data directly to RMA (Remote memory access) operations, resulting in poor performance in scenarios with good locality. GAM is an RDMA-based distributed memory system and also incorporates a local cache to absorb remote data access. However, its design that aims at applications with bulk `Read`/`Write` results in significant performance overhead in data access path. Gemini is a specialized distributed graph analytics engine that has many optimizations targeted at graph analytics workload.

6.2 Micro Benchmark

We first evaluate the total throughput of sequentially accessing data structures by utilizing the `Read`, `Write`, and `Operate` APIs. We compare `Operate` interface in DArray with the `Atomic` interface in GAM, which results in suboptimal performance due to its exclusive ownership.

Workload. We allocate a global array that spans multiple nodes, with each element of 8 bytes in size. The array size increases linearly with the number of nodes, specifically by 0.78 GB per node. Each thread on a node sequentially accesses the entire global array with an 8-byte granularity.

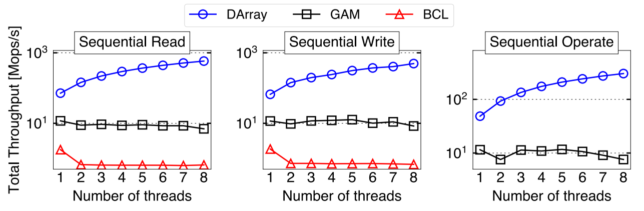

Figure 12: Comparison of sequential (a) Read (b) Write (c) Operate request throughput (Mops/s) with the increase of threads on three nodes.

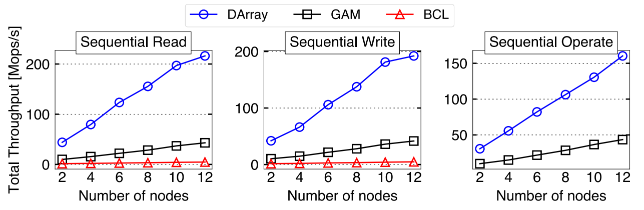

Figure 13: Comparison of sequential (a) Read (b) Write (c) Operate request throughput (Mops/s) with the increase of nodes. Use one thread per node.

Intra-node scalability. Efficient utilization of multi-core resources of servers is crucial in parallel and distributed applications. Therefore, in this micro benchmark, we increase the number of threads on three nodes synchronously to evaluate the intra-node scalability of these systems. The results, shown in Figure 12, indicate that DArray consistently outperforms both GAM and BCL. In such a sequential access scenario, the lack of local cache in BCL results in an average access latency equivalent to the round-trip of network communication, making its total throughput the lowest of these systems. Although we expect BCL's throughput to scale with an increasing number of threads, its scalability is hindered by issues with RMA operations in MPI [13]. Although the use of local cache significantly improves GAM's performance compared to BCL, its high-cost data access path still results in a significant performance gap relative to DArray's low-overhead abstraction. This gap only increases with a growing number of threads.

Inter-node scalability. The significance of inter-node scalability as depicted in Figure 13 cannot be ignored. DArray outperforms GAM and BCL in terms of both performance and scalability. DArray has better inter-node scalability due to its efficient runtime and optimized RDMA communication layer. As the number of nodes increases, DArray has scalability ratios of 0.82, 0.76, and 0.87 for the three operations (`Read`, `Write`, and `Operate`) respectively, which is higher than that of GAM's 0.72, 0.68, and 0.73, and BCL's 0.52 and 0.52 for the same set of operations.

6.3 Optimization Techniques

Two micro benchmarks are utilized to demonstrate the efficacy of these optimization techniques, highlighting the performance improvements brought forth by the "Operate" and "Pin" interfaces.

Operate interface. The workload consists of `write_add` operations that follow a Zipfian distribution of skewness 0.99. To achieve the same semantics without the Operate interface, we would rely on WLock+Read+Write. Evaluation results are shown in Figure 14. The implementation that utilizes the Operate interface exhibits strong scalability and sustains consistent operation latency with

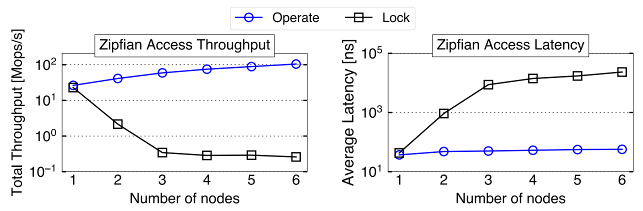

Figure 14: The (a) throughput and (b) latency of zipfian access on a global array using different interfaces. Use one thread per node.

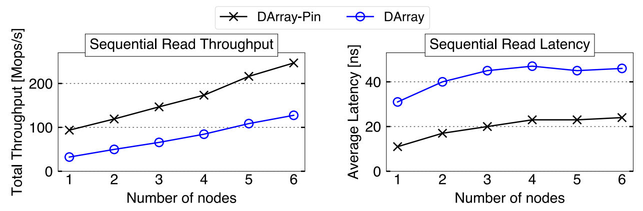

Figure 15: Comparison of DArray and DArray-Pin's sequential 8-byte read performance. Use one thread per node.

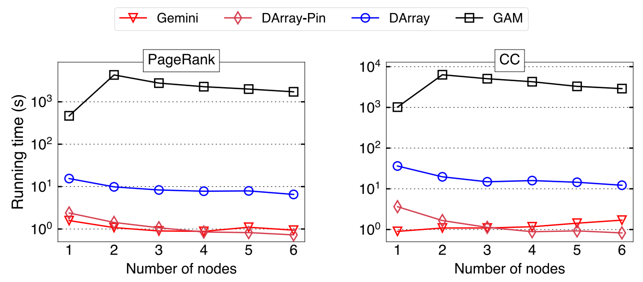

Figure 16: Running time of two graph applications on rMat24. Use all available cores on each node.

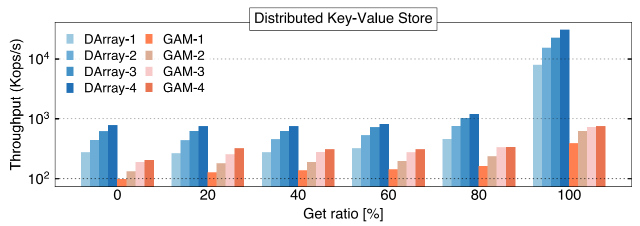

Figure 17: Total throughput (Kops/s) of KVS with varing number of threads and get ratio. The "get ratio" represents the proportion of "get" requests in relation to the total number of "get" and "put" requests.

an increase in the number of nodes. In contrast, the performance of the lock-based implementation is suboptimal, which includes a reduction in overall throughput and a significant increase in data access latency as the number of nodes increases. This outcome is mainly due to the lock-based scheme's exclusive ownership, which causes severe contention in multi-node systems, ultimately affecting performance significantly.

Pin interface. We compare DArray and DArray-Pin in a sequential read scenario to evaluate the performance improvements brought by the Pin interface. As shown in Figure 15, DArray-Pin outperforms DArray by 1.8x to 2.9x in terms of throughput. This is primarily due to the capability to hold references explicitly at the chunk granularity in sequential access scenarios, significantly reducing the need for atomic variable read/write with the use of the Pin interface.

6.4 Graph Analytics

We utilized the array abstractions provided by DArray and GAM to port Polymer [22], a single-machine graph analytics engine, to distributed ones.

Graph applications. Two fundamental algorithms in graph analytics, PageRank (PR), and Connected Components (CC) are implemented utilizing these engines.

Input graph. The input graph, rMat24, containing 2^{24} vertices and 2^{26} edges, is generated by the RMAT generator [6] in Graph500 [2].

Evaluation results of graph applications are shown in Figure 16, revealing that DArray outperforms GAM by two orders of magnitude. This is primarily due to two reasons: 1. Graph applications necessitate a low-overhead abstraction to enhance their performance due to their fine-grained data access. 2. DArray's lock-free design and Operate interface meet the need for high parallelism demanded by graph applications.

We also compare DArray-Pin with Gemini, a specialized distributed graph analytics engine. Despite optimized with Pin interface, abstraction overhead is not negligible due to inevitable branch instructions, resulting in inferior performance of DArray-Pin on a single node compared with Gemini. However, as the number of nodes increases, DArray-Pin eventually outperforms Gemini with speedups of 1.3x on PageRank and 2.1x on Connected Components. This is primarily due to DArray's Operate interface and efficient layered design, which enables computation and communication overlapping, network latency masking with prefetching mechanisms. Additionally, DArray-Pin achieves better scalability ratios, with 0.55 and 0.74 on PageRank and CC, respectively, compared to Gemini's inferior scalability ratios of only 0.28 and 0.09.

6.5 Distributed Key-Value Store

Distributed key-value store is an essential component of distributed systems. GAM has a KVS implementation that is similar to DArray-based KVS, enabling us to compare the two. YCSB benchmarks are conducted on six nodes with a Zipfian distribution parameter of 0.99, which is the default value. The results are shown in Figure 17.

DArray-based KVS consistently outperforms GAM-based KVS in all scenarios primarily due to its low-overhead abstraction, efficient runtime, and optimized RDMA communication layer. With a get ratio of 100%, DArray-based KVS outperforms GAM-based KVS by a factor of 20 to 41. Despite the high contention from PUT requests, DArray-based KVS outperforms GAM-based KVS by a factor of 2 to 3.8. Furthermore, DArray-based KVS demonstrates better intra-node scalability due to its lock-free data access path. DArray-based KVS exhibits a scalability ratio of 0.63-0.96, whereas GAM-based KVS only has a scalability ratio of 0.48-0.64.

6.6 Limitations

Poor locality. To investigate the performance of DArray with poor data locality, we perform a uniform random access over the global array and compare the average access latency of different systems. The evaluation results are shown in Figure 18. When accessing a single-node data structure without network communication, DArray has comparable performance to BCL and outperforms

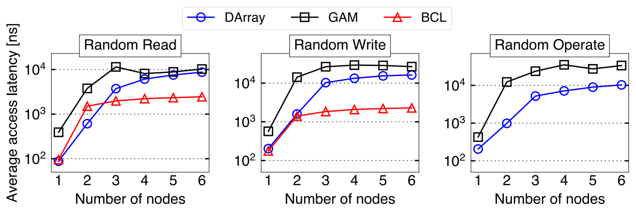

Figure 18: Comparison of random (a) Read (b) Write (c) Operate request latency (ns) with the increase of nodes. Use one thread per node.

GAM due to the lock-free data access path. However, as the number of nodes increases, the average access latency of DArray and GAM increases, while the average access latency of BCL remains roughly stable (approximately 2µs, which is the round-trip latency of RDMA). This is mainly due to the overhead of the cache coherence protocol. When the local cache is crowded, it needs to evict data in the cache before accepting new data. Additionally, we observe that the write latency for random access is higher than the read latency due to some data contention among different nodes.

7 RELATED WORK

RDMA-Based distributed memory systems. FaRM [10] utilizes RDMA to create a shared address space for the memory of all machines in a cluster, and provides developers with interfaces for memory allocation, free, and read/write. FaRM also offers interfaces for transactions, leveraging the atomicity of cacheline updates. GAM [5] implements a distributed cache coherence protocol based on RDMA to ensure data consistency. Similar to FaRM, GAM provides interfaces for memory allocation, free, and read/write operations, as well as additional interfaces with lock semantics.

Distributed memory systems using programmable switches. The emergence of programmable switches has propelled research on implementing distributed memory systems using them, since programmable switches enable in-network computation. Concordia [20] and MIND [18] delegate cache coherence protocol maintenance to programmable switches, allowing them to quickly handle coherence requests. However, these systems require specialized hardware that has not yet been widely used.

Distributed data structures. DASH [12] is a distributed data structure that supports Partitioned Global Address Space (PGAS). It provides the abstraction of an array to developers, but is not optimized with caching and does not offer cache coherence ensurance. BCL [4] implements a richer set of distributed data structures, including queues, hash tables, arrays, and sets. However, like DASH, it incurs excessive network requests for each access to remote data, which makes it unsuitable for applications with good locality.

8 CONCLUSION

This paper presents DArray, a high performance RDMA-based distributed memory system. DArray provides an abstraction of a global array and provides a rich set of optimized interfaces with object granularity. With the objective of high performance, DArray is designed with a coherent distributed cache and a lock-free data access path. Furthermore, the interfaces and cache coherence protocol of DArray are extended to better support applications with commutative and associative data operations. Two distributed applications, a graph engine and a distributed key-value store, are built to demonstrate the versatility and efficacy of these techniques. The results demonstrate that DArray performs well: it consistently shows performance advantages over GAM [5] and BCL [4].

ACKNOWLEDGMENTS

This work was supported in part by the National Key Research & Development Program of China (No. 2022YFB4500700), the National Natural Science Foundation of China (No. 62272291), the Fundamental Research Funds for the Central Universities, as well as research grants from Huawei Technologies.

REFERENCES

[1] 2022. Intel Memory Latency Checker. https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html.
[2] 2023. Graph500. http://www.graph500.org/
[3] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proc. WWW*. 107–117.
[4] Benjamin Brock, Aydın Buluç, and Katherine Yelick. 2019. BCL: A Cross-Platform Distributed Data Structures Library. In *Proc. ICPP*. Article 102, 10 pages.
[5] Qingchao Cai, Wentian Guo, Hao Zhang, et al. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB* 11, 11 (jul 2018), 1604–1617.
[6] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *Proc. SIAM*. 442–446.
[7] Rong Chen, Jiaxin Shi, Yanzhe Chen, et al. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proc. ACM EuroSys*. Article 1, 15 pages.
[8] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*. 137–150.
[9] Peter J. Denning. 2005. The Locality Principle. *Commun. ACM* 48, 7 (jul 2005), 19–24.
[10] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, et al. 2014. FaRM: Fast Remote Memory. In *Proc. USENIX NSDI*. 401–414.
[11] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.
[12] Karl Fuerlinger, Tobias Fuchs, and Roger Kowalewski. 2016. DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms. In *Proc. IEEE HPCC/SmartCity/DSS*. 983–990.
[13] Nathan Hjelm, Matthew G. F. Dosanjh, Ryan E. Grant, et al. 2018. Improving MPI Multi-Threaded RMA Communication Performance. In *Proc. ICPP*. Article 58, 11 pages.
[14] Yang Hong, Yang Zheng, Fan Yang, et al. 2019. Scaling out numa-aware applications with rdma-based distributed shared memory. *Journal of Computer Science and Technology* 34 (2019), 94–112.
[15] Michael Isard, Mihai Budiu, Yuan Yu, et al. 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proc. ACM Eurosys*. 59–72.
[16] Tor E. Jeremiassen and Susan J. Eggers. 1995. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In *Proc. ACM PPoPP*. 179–188.
[17] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, et al. 2015. Turning Centralized Coherence and Distributed Critical-Section Execution on Their Head: A New Approach for Scalable Distributed Shared Memory. In *Proc. ACM HPDC*. 3–14.
[18] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, et al. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proc. ACM SOSP*. 488–504.
[19] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, et al. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proc. ACM SIGMOD*. 135–146.
[20] Qing Wang, Youyou Lu, Erci Xu, et al. 2021. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *Proc. USENIX FAST*. 277–292.
[21] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, et al. 2010. Spark: Cluster Computing with Working Sets. In *Proc. USENIX HotCloud*. 10.
[22] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-Aware Graph-Structured Analytics. In *Proc. PPoPP*. 183–193.
[23] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, et al. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proc. OSDI*. 301–316.