Efficient Runtime Support for a Partitioned Global Logical Address Space

D. Brian Larkins
Rhodes College
Memphis, TN
larkinsb@rhodes.edu

John Snyder
Rhodes College
Memphis, TN
snyjm-18@rhodes.edu

James Dinan
Intel Corporation
Hudson, MA
james.dinan@intel.com

ABSTRACT

Many HPC applications have successfully applied Partitioned Global Address Space (PGAS) parallel programming models to efficiently manage shared data that is distributed across multiple nodes in a distributed memory system. However, while the flat addressing model provided by PGAS systems is effective for regular array data, it renders such systems difficult to use with loosely-structured or sparse data. This work proposes a logically addressed PGLAS model that naturally supports a variety of data models through the automatic mapping of an application-defined key space onto the underlying distributed memory system. We present an efficient implementation of the PGLAS model in the form of a parallel distributed hash table (PDHT) and demonstrate that this model is amenable to offloading using the Portals 4 network programming interface. We demonstrate the effectiveness of PDHT using representative applications from the computational chemistry and genomics domains. Results indicate that PGLAS models such as PDHT provide a promising new method for parallelizing applications with non-regular data.

ACM Reference Format:
D. Brian Larkins, John Snyder, and James Dinan. 2018. Efficient Runtime Support for a Partitioned Global Logical Address Space. In *Proceedings of International Conference on Parallel Processing (ICPP'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Partitioned Global Address Space (PGAS) parallel programming models, such as OpenSHMEM [27], Unified Parallel C [37], Fortran 2008 Coarrays [32, 33], and the MPI Remote Memory Access (RMA) interface MPI [22], have become popular as a means for supporting distributed, shared global data. In particular, PGAS models have been shown to be especially effective when used to store dense arrays and several models have been designed to support this type of data [8, 14, 24, 25].

While PGAS models are commonly used with regularly structured data, they have had limited success with irregular and sparse data. Such data structures are accessed indirectly and an additional indexing operation is needed to resolve the location of a data element. Indexing structures grow proportional to the data size; thus, for the large data volumes that necessitate a PGAS solution, these structures must also be distributed. As a result, resolving the index of a data item within the global address space is not possible using only local information.

Remote direct memory access (RDMA) read, write, and atomic update operations are commonly supported by the high-speed fabrics used to construct HPC systems and these operations are used to accelerate remote access operations performed by PGAS models. RDMA operations map well to remote array access operations, since the locations to be accessed can be determined at the initiator of the operation. However, the indirect addressing required to remotely access sparse or irregularly structured data presents challenges to existing RDMA interfaces and can result in significant overheads.

The Portals 4.1 network programming interface [3] provides a well-known interconnect programming model that is designed to permit efficient hardware offload of communication processing. For example, the Bull* BXI* interconnect [5] provides a hardware accelerated implementation of the Portals 4.1 interface and the Cray* Seastar* interconnect [4] provided a hardware accelerated implementation of the Portals 3.0 interface. Parallel programs written using programming models that are built on the Portals interface will be able to take advantage of network offload hardware. The problem, of course, is to map application-level data and task abstractions onto programming constructs that are aligned with the communication operations supported by low-level communications systems.

Portals is designed as a low-level abstraction layer for the network interface hardware. The Portals communication operations operations are meant to support the efficient implementation of high-performance programming interfaces and runtime systems that provide either message-passing semantics (such as MPI), or one-sided, partitioned global address (PGAS) support. For message-passing systems, Portals provides a *matching interface* that allows communication middleware to robustly support a two-sided (matched sends and receives) communication model. Alternatively, Portals also provides a *non-matching interface* that sheds much of the complexity in matching messages for efficient handling of one-sided communication operations found in PGAS systems and languages.

Rather than considering the matching and non-matching interfaces as the basis for higher-level communication systems, we have developed a system that provides a distributed key/value storage that use Portals primitives in a novel manner, leading towards a hardware accelerated hash table system providing a PGLAS programming model. In this paper, we describe the implementation of a parallel distributed hash table (PDHT) that is implemented directly using the Portals programming interface. This work is based on the following insights: (1) implementing data structures using Portals leads to an

ICPP'18, August 2018, Eugene, OR USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

opportunity to leverage efficient, hardware accelerated implementations of Portals and (2) that operations on a hash table can be readily mapped to operations within Portals that would be difficult to express exclusively within an MPI-like system or a PGAS system.

This work makes the following contributions: We identify the logically-addressed PGLAS paradigm as a novel extension to existing PGAS programming concepts, which improves support for sparse and loosely structured data. We demonstrate the utility of this model through two representative scientific applications, MADNESS and Meraculous, from the computational chemistry and genomics domains, respectively. We describe a new PGLAS model, PDHT, a one-sided distributed hash table that is implemented using a novel application of Portals matching interface operations. Lastly, we provide an experimental validation, indicating that this approach can yield significant performance improvements for these classes of applications.

2 BACKGROUND

This work focuses on an efficient implementation of a runtime system that provides a partitioned, global logical address space (PGLAS). Similar in spirit to traditional PGAS data models, the PGLAS distinguishes itself by logically addressing data elements with a *key*, which is meaningful to the application. Keys are mapped to a unique location in the partitioned global address space. The application may iterate over local keys or get locality information for the logically addressed element in order to support owner-compute work distribution. Rather than application-specific keys for distributed data, conventional PGAS data models rely on direct remote addresses (e.g. OpenSHMEM and UPC), an offset into a remote buffer (MPI), or an index into an array (Fortran Coarrays).

While PGLAS and PGAS models differ in addressing and layout, PGLAS shares a one-sided communication model and asynchronous data access common to existing PGAS approaches. Data is accessed using one-sided get, put, and atomic operations. The PGLAS model is a member of the larger family of distributed key/value stores where the organization and access of data follow a PGAS model.

There are several important distinctions between the traditional global memory addressing and management under the PGAS model versus the PGLAS approach. PGAS models typically require global management of the addresses space. In the OpenSHMEM model, for example, the globally accessible address space must be the same on every process. This requires global coordination or the use of collectives in order to effectively manage shared memory regions. In contrast, the PGLAS model introduces an indirection layer between the logical and physical global addresses. This avoids the overhead common to PGAS systems and is used to relax the memory management semantics of such systems even further. This indirection fosters more dynamic global memory usage, specifically with the support for dynamic insertions and removals of key/value pairs in the shared global address space.

Achieving high-performance with PGLAS is only possible with an efficient mechanism for translating logical addresses (keys) to physical addresses within a PGAS system. One-sided, asynchronous PGAS operations can be used to resolve logical addresses, but it is difficult to do efficiently. Resolution may require several communication operations to query a remote translation directory, resulting in high overheads [23].

MPI provides application-specified tags to match up specific pairs of send/receive operations. MPI tags could be used in place of more general PGLAS keys, however there are limitations with this approach. First, MPI requires ordered message processing, whereas both PGLAS and PGAS systems adopt unordered-by-default communication models. Next, tags are intended to be used as a mechanism to specify the matching of specific send/receive messages, not to implement a persistent key/value store. Further, tagged receive operations are consumed by incoming messages, whereas the PGLAS model permits the key/value association to persist until explicitly released by the application. Lastly, tagged MPI requests for data objects that do not exist will result in the request being deferred to the unexpected message list, rather than a negative acknowledgement. While this is reasonable for a two-sided communication model, explicitly rejecting PGLAS requests for non-existent data objects is prudent.

2.1 Distributed Sparse Data

We see the PGLAS model as a tool for dealing sparsity within HPC problem domains. Sparse data structures are commonly used in a number of important HPC applications, which can create challenges in obtaining high performance. Matrix and tensor representations frequently have to contend with sparse data, relying on specialized formats such as compressed sparse row (CSR) and variants. Other problems, such as grid/mesh solvers or N-body simulation are naturally expressed with spatial decomposition data structures such as quadtrees and octrees.

In these instances, PGLAS provides a natural mapping between an application's inherent indexing system and the corresponding distributed data. Global distributed key/value stores make it straightforward to deal with sparse data and are helpful in avoiding the complications of parallelizing these classes of applications.

2.2 Related Work

Logically addressing data across memories in a distributed system has been explored with other models. Distributed, shared key/value stores are an intrinsic part of the Piccolo programming model [31]. The Linda programming model [1] is built around tuple spaces that allow data to be inserted, read, and removed from global memory. Linda data items are identified through a user-supplied tuple. Such models are of renewed interest in the context of resilience [38].

Distributed object models, such as Orca [2], reference shared data through graphs rather than direct pointers. In the Open Community Runtime (OCR) [26], data blocks (DBs) are identified by a globally unique ID, which is assigned by the OCR runtime system and used to portably reference a DB. In the MPI-3 RMA model [22], windows allowing remote access to the private memory of a process can be created. However, window creation is collective and each process must maintain a table of all window handles. Finally, migratable object models, such as Charm++ [12], support a variety of methods for representing and querying object collections.

Hash tables are also commonly used to store sparse or loosely structured data [6, 28, 34] and their implementation has been studied

in the context of MPI and popular PGAS models [10, 19, 20, 35, 39, 40] as well as in the context of RDMA offload [13, 21, 35]. Others have studied the applicability of programmable NICs to efficiently implement key-value stores [18]. In addition, PGAS-like hierarchical approaches to managing sparse spatial decomposition have been proposed [17] along with runtime methods to manage this data and exploit spatial locality [15].

3 IMPLEMENTATION

To realize our PGLAS, we have implemented a parallel distributed key/value store, PDHT. A novel aspect of this system is its implementation directly on top of the Portals4 network interface [3]. The Portals framework provides several low-level data abstractions that map nicely onto the PGLAS operations supported by PDHT.

PDHT adopts a one-sided communication model which operates with PGAS-like semantics, providing put, get, and atomic operations. By implementing PDHT directly with the Portals interface, we can take advantage of message-passing features that permit an efficient, hardware-offload friendly version of a PGLAS. Describing some of the primary abstractions with the Portals network model is helpful to better understand the PDHT implementation.

3.1 Portals Networking Interface

The Portals interface provides low-level data abstractions that can be used to implement both PGAS and message-passing run-times. PDHT relies on a one-sided communication model that aims to avoid processor involvement on the target node. Portals supports messaging via a *matching interface* model intended to support MPI (i.e. two-sided matched send/receive pairs). It also provides a lightweight *non-matching* interface that is intended to support one-sided communications without the need for the matching and ordering semantics required by MPI. Our implementation uses a novel application of the Portals matching interface to provide a one-sided model that is the basis of the PDHT PGLAS.

The Portals matching interface was designed to support systems that provide a two-sided messaging model with guarantees about message ordering. The implementation strategy that underlies PDHT is to separate these two concerns. PDHT utilizes the matching interface operations to fulfill put and get operations but makes no guarantees about the interleaving of requests. We review the primary concepts in the Portals interface to provide clarity for discussing the implementation of PDHT.

Portals defines two distinct roles for communication operations. The *initiator* role is assumed by the process that issues a Portals put, get, or atomic operation. The *target* role is assumed by the process that receives these operation requests and is responsible for performing the operation or sending the requested data.

The Portals *network interface* (NI) data structure is a per-process abstraction of a physical network interface and is configured as a matching or non-matching interface. The NI is associated with both initiator and target roles. Initiator processes define a *memory descriptor* (MD) – a region of memory used for the put/get/atomic operations as well as ancillary structures used to track communication events (such as completion) for these operations.

Target processing is expressed with respect to a *portal table*. Each NI has one portal table containing multiple entries that separate and

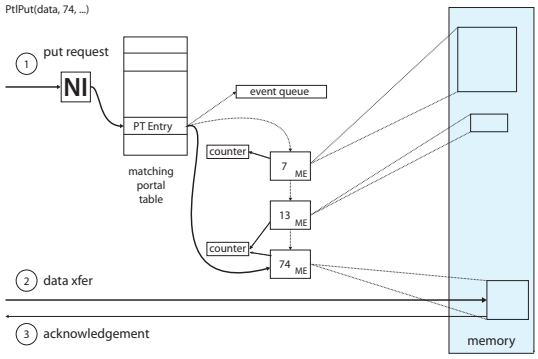

Figure 1: Target-side data abstractions used for a Portals matching interface `put` operation

classify communication channels between endpoints. Each *portal table entry* (PTE) is identified by a unique index into the Portal table. The index for a specific PTE is a required parameter of all Portals communication operations. Each PTE maintains a list of memory regions that are valid for communication operations as well as an event queue, which is used to track target-side asynchronous notifications. Some completion information and notifications may only be visible to the initiator and not the target, or vice versa.

A PTE using the matching interface maintains a list of match-list entries (ME). Each ME specifies a set of matching criteria and a memory region associated with the entry. If the match criteria are satisfied, the communication operation commences working on corresponding memory region. Matching criteria consist of multiple parameters, including a 64-bit *match bits* field that must be an exact match in order for an incoming request to to proceed with the given ME. For example, a process may post an `MPI_Recv()`, which creates a new ME on the target PTE with a specific match bits value. Later, another process would issue an `MPI_Send()` communication request with the same match bits that correspond to the previous receive operation.

Portals also provides *event queues* and *event counters* that can be used to track data movement in and out of memory regions related to Portals communication operations. Events and counters also log failure events and additional information. Counting events are lightweight operations and only record the success or failure of a given operation. Full events incur more overhead but contain detailed information about the event and the corresponding communication operation. Event queues may be associated with an MD (initiator-side) or a PTE (target-side).

Event counters are also able to track both initiator and target events. In contrast to target-side event queues, an event counter may be associated with a single entry in the match list (an ME), rather that for an entire PTE. PDHT typically uses lightweight counting events to track successful completions and full events to track specific failures. Detailed descriptions of how event queues and counters are used within PDHT is discussed below.

Figure 1 shows a visual representation of the target-side structures used in an example `PtlPut()` operation using the matching interface. We assume that the initiator has issued a `PtlPut()` operation with some data, a PTE index and the match bits set to 74. Upon receiving the request, the target process searches the match list for the given PTE until a 74 is found or the end of the list is reached. Once a match has been found the payload data of the `PtlPut` is copied into the memory region associated with the ME. The completion of this operation may cause an event to be appended to the PTE event queue or the ME success counter to be incremented. Lastly, an acknowledgement message is sent back to the initiator, possibly generating another full or counting event associated with the MD that was used to initiate the `PtlPut` operation. Note that the match list is searched in sequential order, preserving message-ordering semantics typically required by MPI implementations.

In a two-sided model, a send request may arrive at the target prior to the receive being posted. In addition to the main (priority) match list, Portals maintains a secondary overflow match list for unexpected messages. The overflow list is searched automatically if a matching entry is not found on the priority match list. The overflow list may also be searched explicitly by the target-side process (e.g. to match an earlier send to a posted receive).

3.2 PDHT Implementation

Hash tables and key/value stores can be implemented with a myriad of different techniques but are traditionally implemented sequentially as an array of pointers to objects. Parallel implementations typically distribute entries across nodes. In contrast to sequential versions, the distributed store must hold the entire value object, rather than just holding a reference to the value object (to avoid another communication).

Our implementation of the PGLAS data model, PDHT, provides a distributed key/value store that is accessed using asynchronous one-sided *insert, read, write,* and *atomic update* operations. PDHT uses a traditional hash function[29] to map an arbitrary user-supplied key to a process rank and 64-bit hash value. Each key is mapped to a deterministic physical location within the distributed key/value store.

A PDHT storage structure is represented as a distributed array of objects, with partitions on all processes. Upon the creation of a new PDHT store, two portal table entries (PTEs) are allocated. The first PTE is used as the *pending* match list, which is used solely for insert operations. The second PTE is used for the *active* match list, and is used for all other operations.

The pending match list is pre-populated with a number of entries that are configured to match any incoming communication requests (i.e. wildcard entries). Each match list entry (ME) in the pending list is mapped to a distinct memory region located in the local portion of the distributed object array. All MEs on the pending match list are marked as use-once and are automatically removed from the list after being written to. In contrast, MEs on the active list may either be marked as use-once or persistent, depending on the needs of the application.

Every live element in the key/value store has a corresponding active match list entry. The ME has a reference to the location in the distributed object array and uses the 64-bit hash value produced by the hash function as the matching value, (*match bits*), in the match list. By using the hashed value of the key for the match bits field in the ME, the lookup and communication corresponding to the read operation can be performed entirely by the Portals implementation.

3.3 Inserting Objects

To insert a new element into the hash table, the process first hashes the key, yielding both the target processor rank and the 64-bit match bits field used for the value. Local insertions can be handled as a special-case, by copying the entry into an available region in the table on the local process and appending a match list entry onto the active PTE with the match bits set to the value returned by the hash function. In this case, the local process acts as both the initiator and target processes.

If the key hashes to a remote process, the initiator process performs a one-sided Portals put operation onto the pending PTE of the target rank. This consumes a wildcard entry on the match list. This process is demonstrated in Figure 2. The initial state is shown in Fig. 2a, with wildcard entries on the pending match list and available key/value pairs on the active match list. After the put operation completes, the hashed key match bits are set on the consumed wildcard ME (K_4) and the value data (V_4) is stored into the memory region defined by the match list entry, as can bee seen in Fig. 2b.

The value object data is communicated and stored into the memory region defined by the match list entry. As seen in Figure 2b, the table entry resides on the owner process, but the match-list entry is still resident on the pending match list, rather than the active one. To complete the put operation, the inserted entry must have a corresponding entry on the active match list, with the match bits set to the hashed key string. Since the insert operation consumed a wildcard entry on the pending PTE, it must also be replaced with a new empty ME.

The final state, represented in Fig. 2c, shows that the K_4 entry has been appended to the active match list and is available to be accessed by other processes. Entries are only accessible once they have been appended to the active match list. The insert operation returns on the initiator side once it receives an acknowledgement that the target has placed the new entry onto the pending match list.

A dedicated progress thread monitors the event queue on the pending match list and handles the migration of the entry from the pending list to the active list. The progress thread blocks on the event queue corresponding to the pending PTE and is notified by Portals whenever a new insert request arrives. The progress thread replenishes the consumed wildcard ME on the pending list and appends a new ME to the active match list with the appropriate match bits set, as can be seen in Fig. 2c. Any subsequent data access operations will match against the entry in the active list.

It is possible for insert operations to outpace the progress thread's ability to replenish entries on the pending match list. Ideally, the pending match list size should be large enough to buffer incoming requests, but not as large as the active lists. PDHT implements a flow control protocol using the Portals flow control primitives and event notifications to pause insert operations until there there are valid wildcard entries on the pending match list. Invoking flow control does incur overhead and may cause multiple communication attempts. While this increases the time to insert a new entry under a

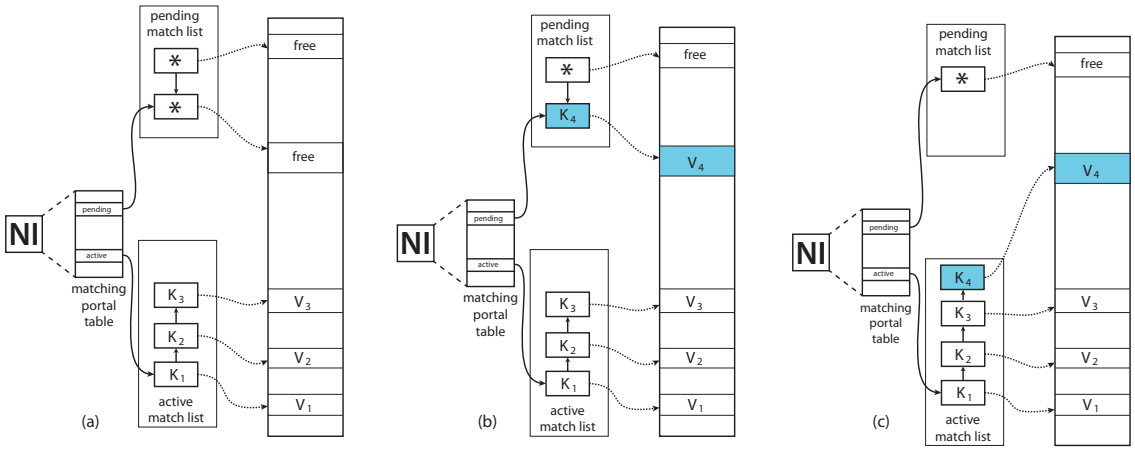

Figure 2: Sequence of target-side actions performed during a PDHT `insert()` operation.

flow control scenario, it avoids the application programmer burden of selecting an optimal pending queue size.

3.4 Accessing Objects

In contrast to the complexity of adding new items to the PDHT, other operations, such as reading or updating existing entries, are straightforward. PDHT supports get, update, and atomic update operations on key/value pairs. When an operation is requested, the key is hashed to yield the rank and match bits value. The system then simply invokes the corresponding Portals get, put, or atomic operation on the active match list with the hashed match bits value. Portals performs the operation and updates an event counter that increments a success or failure counter. In the case of a failure, PDHT discriminates a "not found" error from other networking troubles.

Beyond using read/insert/update operations for working on distributed data, PDHT also provides an interface to iterate over active entries on the local process. This functionality allows for owner-compute style programming idioms and allows for communication-free access to global data entities. Remote fetch or update operations are not synchronized against local iteration. Overlapping both models is trivial in a read-only phase but may require external synchronization when mixing models in general.

3.5 Additional Features

Because PDHT is built directly atop the Portals interface, normal programming features such as collective calls are missing. While PDHT can integrate seamlessly with a higher-level parallel programming framework such as MPI or UPC, there is no requirement for a PDHT application to use such a system. As a result, PDHT has also been extended to provide rudimentary collective operations such as broadcast, reduce, and all-reduce functions for a limited set of common datatypes.

PDHT also provides barrier and fence operations for synchronizing processes. The barrier operation works in the conventional sense, blocking all processes until the barrier is reached. The fence blocks progress until all outstanding communication events invoked prior the fence have been completed on the target process. The PDHT fence operation is implemented as a collective call, which iteratively compares the number of outstanding communication operations against the number of completed operations – the fence terminates when the values match.

The runtime also supports global, atomic counters. These counters are typically used for coarse-grained dynamic load balancing. Global counters are implemented with lightweight Portals atomic operations. The runtime allocates two additional Portal table entries (PTEs) to support the collective, synchronization, and counter operations.

3.6 Performance Improvements

3.6.1 Progress Thread Elimination. A progress thread is a viable approach for handling insertion requests but requires target-side processing by both Portals and PDHT. Portals provides an experimental feature that allows for a triggered match list append operation to occur when an event counter exceeds a specified threshold value. We can eliminate the progress thread altogether by utilizing these *triggered append* operations. In this approach, every PDHT key/-value match list and table entry is associated with a Portals event counter, initially zero. When the wildcard MEs are created on the pending list, triggered append operations are also setup to trigger when the wildcard ME is written to. The responsibility for moving key/value pairs from the pending list over to the active list is delegated to the Portals layer (which may be supported with hardware), rather than a separate user-level progress thread.

One issue with this approach is that Portals requires the match bits for the appended entry to be specified at setup time, rather than when the trigger is executed. Within PDHT, the trigger is set when the wildcard entry is added to the pending match list, but the active ME's match bits are unknown until a new entry is placed via a `PtlPut()` on the pending list. This issue does not impact the current reference implementations of either PDHT or Portals. These issues could be resolved with alternative Portals implementations under the current triggered operation semantics by "chaining" multiple triggered operations. Ideally, this additional functionality and semantics should be incorporated into the Portals specification, in order to ensure that these programming models are portable across Portals implementations.

3.6.2 Unordered Matching. PDHT is able to provide efficient get operations by having an ME for every entry in the PDHT key/-value store. This design results in match lists that may contain many thousands of entries. When a get operation is issued, the target side must search the match list for a matching entry and return the value object associated with the requested key. A significant challenge to implementing a large-scale key/value store on top of this approach is that the match list is conventionally implemented as a linked-list, which much be searched upon any message request. If a process is hosting a large number of key/value entries, searching this list may be prohibitively expensive. Traditional uses of the matching interface, such as implementing MPI message matching, would rarely see match list lengths on the scale as may be used in PDHT (one per element) [7].

Indeed, initial results showed that match list length had a critical impact on the access times within PDHT [16]. Our first approach to deal with this issue was to adapt the hash function to map a logical key to a tuple including the rank, match bits, and a new PTE index value. The PDHT implementation is able to distribute key/value objects over multiple Portal table entries. Instead of a single entry with a match list containing 10,000 entries, the system could be configured with 10 PTE match lists, each containing approximately 1,000 entries. While this technique did reduce match list overhead, search latency was still substantial, especially on small scale runs, or with very large datasets.

When used in message-passing libraries, the matching procedure must preserve the ordering of posted communication operations – a behavior that is easily captured with a linked list. However, under the PGLAS approach, no ordering constraint exists and sequentially searching a linked list unnecessarily slows performance.

We have therefore proposed the inclusion of an unordered matching feature on the Portals matching interface and the corresponding patches has been merged into the Portals reference implementation. We implemented this extension by adding a `PTL_PT_MATCH_UNORDERED` option that is specified when creating a new PTE. When the unordered matching option is set, the match list append operation stores a reference to the ME in a lightweight hash table [11]. The search process is also amended to circumvent a list traversal by first checking the hash table for a matching entry. This results in much improved performance with respect to the initial implementation.

3.6.3 Local Match List Searching. Initially, all accesses to distributed value objects were made using `PtlGet()` operations. The Portals reference implementation makes no distinction between local and remote data, which adds substantial overhead for operations on local data.

Every Portals matching PTE contains two message match lists – the unexpected list and priority list. In two-sided communication, such as MPI, an `MPI_Send()` may arrive before the `MPI_Recv()` is posted. Portals exposes API to search the unexpected message list to match the send/recv messages. A similar approach, but to the priority list (i.e. the active match list) could be used to improve access for value objects local to the requesting process.

We added a new priority match list search call to the Portals API and adapted the existing code in Portals to provide this functionality. When the new search function is called, an event is generated containing a pointer to the value object if found, or a failure event otherwise. PDHT get and update operations check the rank component of the tuple returned by the hash function. When the object's rank matches the local process, PDHT calls the search function for local entries rather than calling `PtlGet()` or `PtlPut()`. This optimization led to a dramatic improvement in access to local data.

3.7 Hash Collisions

A hash collision occurs when multiple keys map to an identical rank/-match bits pair. By default, PDHT detects collisions by comparing the requested key value against the key of the retrieved object and reporting the condition to the application. PDHT can also handle collisions transparently within the system by reserving a small number of match bits to use as alias identifiers and probing multiple aliases on the requested key when a collision is detected. However, we have observed that many applications can provide a hash function that is collision-free.

In traditional hash tables, a hashed key, K, is used with modular arithmetic to select a location in an array of elements of size N. The hash function gives a mapping between $K \rightarrow N$ elements. If the size of the key space, $|K|$, is much larger than N, the likelihood of collisions increases. Conventional approaches store a list of objects within each array entry (chaining) to limit the overhead of collisions.

While PDHT also uses an array to hold the collection of hash table entries, the indexing into this array is performed indirectly, by Portals. At initialization, match list entries map to specific table entries. While each table may be of size N, each of the N entries is indexed by the Portals match bits value, using a hash function that maps from $K \rightarrow 2^{64}$. Compared to traditional implementations, $N \ll 2^{64}$, which has the impact of dramatically lowering the probability of a collision, given a reasonable hash function.

4 EXPERIMENTAL RESULTS

Performance results were gathered using a prototype implementation of PDHT running on a version of the Portals reference implementation [30] that was modified to support the features presented in Section 3.6. MPI comparisons were performed on the Comet cluster system at the San Diego Supercomputer Center with access provided through the Extreme Science and Engineering Discovery Environment (XSEDE) program. Comet contains nodes with several configurations; these experiments were conducted a portion of the 1944 nodes with Intel® Xeon® E5 processors, comprising Dell⋆ PowerEdge⋆ C6320 server each with dual 12-core Intel® Xeon® E5-2680 processors and 128 gigabytes of memory. Nodes are connected using a Mellanox⋆ FDR 56Gb/s InfiniBand⋆ fabric. UPC experiments were run on an Advanced Systems Technology cluster at Sandia National Laboratories. The cluster contains 36 compute nodes, constructed with dual 16-core Intel® Haswell® processors and 128 gigabytes of memory. Nodes are connected using a Mellanox⋆ FDR 56Gb/s InfiniBand⋆ fabric.

4.1 Experimental Setup

To measure the effectiveness of implementing PDHT directly on top of Portals, we constructed an MPI version of PDHT. The MPI version implements the PDHT functionality using a companion processes to act as a PDHT server and handle insertion, access, and update

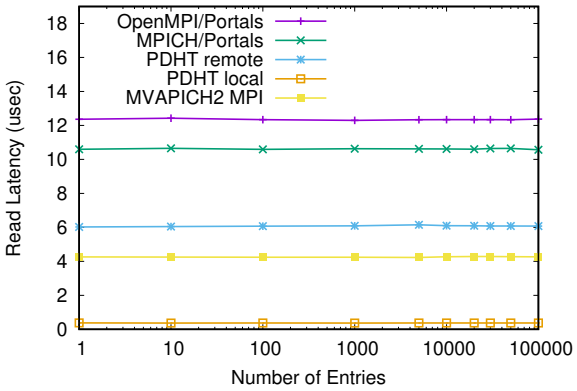

Figure 3: PDHT and MPI Access Latency (μsec)

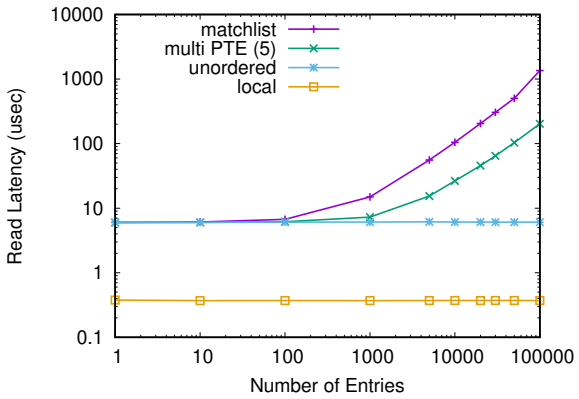

Figure 4: PDHT Access Latencies (μsec)

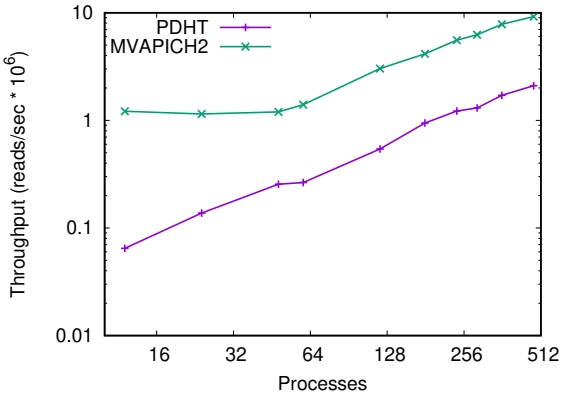

Figure 5: Aggregate system throughput (1K elements).

requests. The MPI PDHT companion process is implemented using MPI send and receive operations. It relies on the same hash function to determine rank and match bits, but internally stores PDHT elements in an efficient hash table on each local process [11], keyed with the same 64-bit value returned by the PDHT hash function.

Experiments run with OpenMPI are using version 3.0.0, which has been configured to use Portals 4 as the message transport layer for all communication operations. The MPICH MPI tests are run with version 3.2.1 using Portals 4 as the `ch3` netmod. The MVAPICH2 tests are run with version 2.1 and the runtime is configured to directly use the InfiniBand* Verbs interface. UPC experiments were run using GCC UPC 5.2.0.1 configured to use Portals 4 for communication.

One of the primary challenges in deriving performance results for this work is that the Portals 4 runtime library was developed as an exploratory framework for modeling next-generation hardware offload network processing. As a result, Portals message processing is performed by the host processor in a Portals-level companion thread and incurs higher overheads than would be expected with a hardware accelerated implementation. Thus, communication performed through the Portals 4 runtime library has not been optimized as extensively as other high-performance network middleware (e.g. MVAPICH2). Where possible in our evaluation, we have measured these overheads and presented comparisons that demonstrate the relative performance potential of PDHT.

4.2 Communication Operation Costs

We start by measuring the read latency of the MPI and Portals PDHT implementations in a two node configuration with one process per node. As can be seen in Figure 3, the latency of the native PDHT implemented directly on Portals is faster than that of the MPI implementations tested with both MPICH and OpenMPI using Portals communication channels. This microbenchmark tests the latency with local hash table sizes of a varying number of items. The MPI PDHT implementation tested with the MVAPICH2 system outperforms all Portals implementations in a raw performance test. We attribute this to the different design goals of the two systems as mentioned above.

Next, we examine the impact of different native PDHT implementation techniques described in the implementation section in Figure 4. Of particular note are the greatly increasing latencies for the initial native PDHT implementation. This is caused by the default $O(N)$ linked-list searching in the Portals reference implementation. When using the unordered matching option, the native PDHT access times are flat $O(1)$ over all match list lengths.

The default version shows growing latencies with respect to match list length. However, by spreading the key/value entries over multiple Portal table entries (i.e. multiple match lists), we see that performance improves, but latencies still grow as more entries are added. Using the unordered matching feature in the current version of Portals, we see that PDHT is able to run with good, flat latencies as the number of local entries increases. Lastly, we see the impact of searching the local match list rather than performing a full `PtlGet()` when fetching entries on the local process rank. This lowers the access latency by an order of magnitude.

As shown in Figure 5, we measure the weak scaling of PDHT read operations on the key/value store by examining the overall throughput of the system. A total of twelve processes are run per node, allowing additional cores to be used for Portals progress threads or MPI companion processes. Each process owns a fixed number of

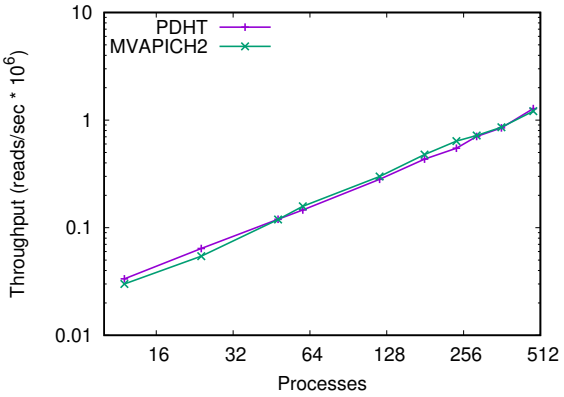

Figure 6: Aggregate system throughput (64K elements).

hash table elements that are read by a neighboring process. With overall system throughput increasing with the number of processes involved, the system performs as expected under weak scaling. We compare PDHT native performance against MPI using the MVAPICH2 runtime.

With smaller entry sizes, we can see that MVAPICH2 performs better than PDHT. This performance gap is largely due to the increased latencies when using a software simulation of Portals on an InfiniBand network. By increasing the size of a key/value entry to 64KB, data transfers become bandwidth bound and the impact of these overheads on performance becomes negligible. Figure 6 shows the convergence in overall system throughput. Portals-based MPI implementations did not scale favorably with either PDHT or optimized MPI; thus we do not report results for these implementations because of performance bugs.

4.3 MADNESS

The MADNESS multiresolution numerical computation framework [36] has been used to solve a variety of quantum chemistry problems at scale. The system utilizes wavelet-based functions implemented using 3 to 6 dimensional spatial representation trees. MADNESS trees represent functions with coefficients stored in either a scaling basis or a wavelet basis. Certain operations on the function tree can only be performed in one or the other basis modes. These spatial decomposition trees are sparse structures, with larger subtrees corresponding to higher-frequency components of the modeled function. PDHT can be used to provide a PGLAS system that represents each tree node in terms of it's spatial coordinates and location (depth) within the tree.

We consider three operations provided by the MADNESS framework. The first two are conversion operations: *compression*, in which scaling coefficients are converted to the wavelet basis, and *reconstruction*, the complementary operation. The last operation is *differentiation*, in which the derivative of a function tree is computed with respect to a single dimension.

The differentiation operation is an irregular operation and is easily expressed using spatial tree coordinates, but is very difficult using traditional message-passing (due to tree irregularity) and PGAS (due

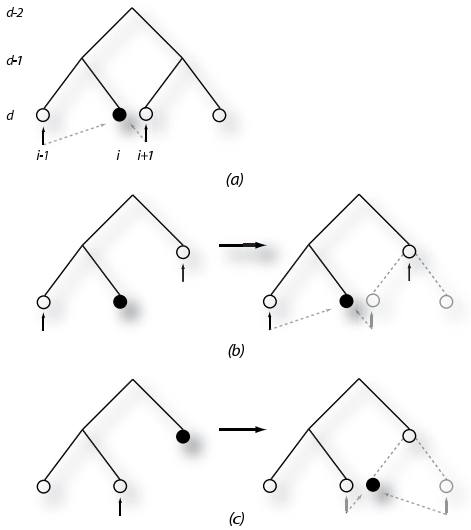

Figure 7: MADNESS differentiation operation

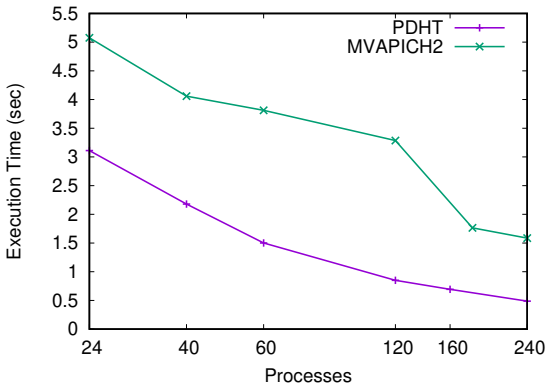

Figure 8: MADNESS compression performance (sec)

to physical addressing requirements). As shown in a Fig. 7a with a 1-D function tree, taking the derivative at node i requires a contraction with coefficients from nodes $i-1$ and $i+1$ at the same level within the tree. If these nodes do not exist, the closest ancestor node must be found and expanded to provide coefficients at the correct level.

Figures 8–10 show the performance of the Portals-based PDHT implementation relative to the MPI implementation.

4.4 Meraculous

The NERSC Meraculous benchmark performs de novo whole genome assembly, reconstructing genomic sequences from overlapping and erroneous fragments produced by a high-throughput sequencer. Meraculous constructs a graph of all overlapping substrings in the input and traverses the graph to discover all linear subgraphs. These subgraphs correspond to contiguous sequences of genomic data. Meraculous is written in UPC and uses a hash table to represent the de Bruijn graph.

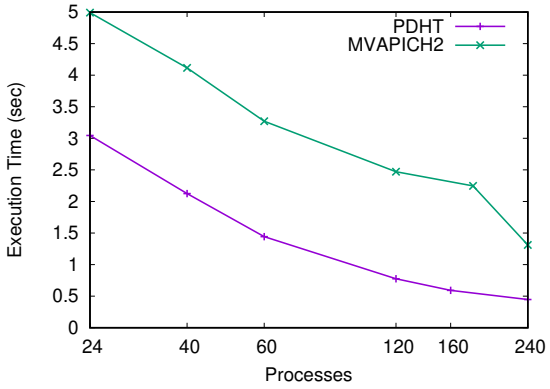

Figure 9: MADNESS reconstruction performance (sec)

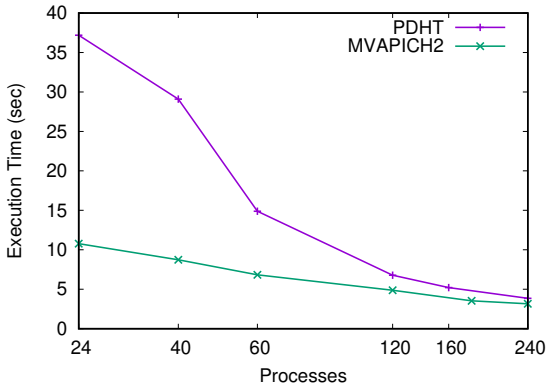

Figure 10: MADNESS differentiation performance (sec)

We have adapted the Meraculous benchmark [9] to store the de Bruijn graph using PDHT instead of a UPC hash table. Both structures utilize the same key structure and creation algorithm. The graph traversal is performed in parallel by iterating through all local entries in the table and walking both the left and right ends of the sequence. The input file for this experiment was the `human-chr14.txt.ufx.bin` data set available as a part of the NERSC benchmark suite. This input set consists of 96 million sequences, each of which must have an ME in Portals. The results of this experiment are shown in Figure 11. At small scale, this consumes a much greater number of Portals resources than is typically supported. PDHT, however, completes without encountering collisions. On the other end, adding processes reduces Portals resources but runs out of work quickly.

processes	UPC	PDHT
10	43.23%	0.0%
20	41.95%	0.0%
40	41.34%	0.0%

Table 1: Meraculous UPC vs PDHT collisions

Collisions are a source of overhead in any distributed hash table. We also explored how the design of the PDHT system can

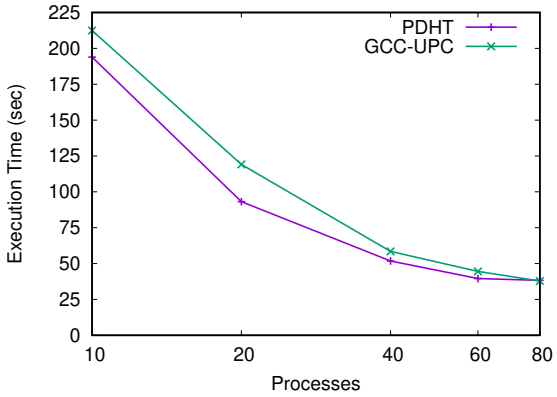

Figure 11: Meraculous UPC vs PDHT performance (sec)

reduce the number of collisions that occur when compared with more traditional direct array-indexed distributed tables. Table 1 provides additional evidence that PDHT outperforms the default UPC array-based implementation of a hash table with respect to collisions. Collisions are counted as any global hash table access that is not satisfied without traversing a portion of the linked-list chain for a given hash table array element. We also noticed similar behavior using a PDHT variant of a benchmark designed to compare UPC, MPI, and OpenSHMEM-based hash tables [20].

5 CONCLUSION

This paper has presented the partitioned global logical address space (PGLAS) programming model and its first realization through the Parallel Distributed Hash Table (PDHT). PDHT system makes novel use of the Portals networking interface by re-purposing its message-matching system to operate as an acceleration mechanism for a global, distributed key/value store.

This work has led to several insights into adapting Portals abstractions onto higher-level programming models. In particular, the separation of the message matching mechanics from required delivery ordering is key to good performance with PDHT. We see future opportunities for adapting the Portals triggered operations to manage PDHT structures. Lastly, this work has motivated several extensions to the Portals model, including functionality allowing applications to search the priority list when responding to local matching queries.

We envision that a number of applications that rely on sparse methods and irregularly structured data can be readily adapted to the PGLAS parallel programming model. We identified two representative applications from the computational chemistry and genomics domains and applied PDHT as a system for managing the sparse spatial and loosely structured data arising in these domains. Experimental results are promising and indicate that Portals can provide efficient and productive support for such data structures.

ACKNOWLEDGMENTS

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

We would like to acknowledge Ryan Grant and Todd Kordenbrock at Sandia National Laboratories for their help with Portals-related issues and the use of Sandia facilities as well as Evangelos Georganas at Intel for his assistance with the Meraculous benchmark.

REFERENCES

[1] S. Ahuja, N. Carriero, and D. Gelernter. 1986. Linda and Friends. *IEEE Computer* 19, 8 (Aug. 1986), 26–34.

[2] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. 1992. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Transactions on Software Eng.* 18, 3 (Mar 1992), 190–205.

[3] Brian W. Barrett, Ron Brightwell, Ryan E. Grant, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabe, and Trammell Hudson. 2017. *The Portals 4.1 Network Programming Interface*. Technical Report SAND2017-3825. Sandia National Laboratories.

[4] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson. 2006. SeaStar Interconnect: Balanced Bandwidth for Scalable Performance. *IEEE Micro* 26, 3 (May 2006), 41–57. https://doi.org/10.1109/MM.2006.65

[5] S. Derradji, T. Palfer-Sollier, J. P. Panziera, A. Poudes, and F. W. Atos. 2015. The BXI Interconnect Architecture. In *Proc. IEEE 23rd Annual Symposium on High-Performance Interconnects*. 18–25. https://doi.org/10.1109/HOTI.2015.15

[6] Ciprian Docan, Manish Parashar, and Scott Klasky. 2012. DataSpaces: an interaction and coordination framework forÂăcoupled simulation workflows. *Cluster Computing* 15, 2 (2012), 163–181.

[7] Mario Flajslik, James Dinan, and Keith D. Underwood. 2016. Mitigating MPI Message Matching Misery. In *Proc. 31st Intl. Conf. High Performance Computing (ISC '16)*. 281–299. https://doi.org/10.1007/978-3-319-41321-1_15

[8] Graham D. Fletcher, Michael W. Schmidt, Brett M. Bodec, and Mark S. Gordon. 2000. The Distributed Data Interface in GAMESS. *Computer Physics Communications* 128 (June 2000), 190–200. Issue 1-2.

[9] 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 *Proc. of the Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 437–448. https://doi.org/10.1109/SC.2014.41

[10] R. Gerstenberger, M. Besta, and T. Hoefler. 2013. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, "", 53:1–53:12.

[11] Troy D. Hanson and Arthur O'Dwyer. 2017. uthash User Guide. https://goo.gl/eynYX1. (Aug. 2017).

[12] Laxmikant V. Kale and Sanjeev Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In *Proc. Conf. on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*. 91–108.

[13] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[14] Amir Kamil, Yili Zheng, and Katherine Yelick. 2014. A Local-View Array Library for Partitioned Global Address Space C++ Programs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. ACM, New York, NY, USA, Article 26, 6 pages. https://doi.org/10.1145/2627373.2627378

[15] D. Brian Larkins. 2012. Improving Data Locality for Irregular Partitioned Global Address Space Parallel Programs. In *Proc. 50th Annual Southeast Regional Conference (ACM-SE '12)*. ACM, New York, NY, USA, 280–285. https://doi.org/10.1145/2184512.2184577

[16] D. Brian Larkins and James Dinan. 2016. Extending a Message Passing Runtime to Support Partitioned, Global Logical Address Spaces. In *Proceedings of the First Workshop on Optimization of Communication in HPC (COM-HPC '16)*. IEEE Press, Piscataway, NJ, USA, 11–16. https://doi.org/10.1109/COM-HPC.2016.7

[17] D. B. Larkins, J. Dinan, S. Krishnamoorthy, S. Parthasarathy, A. Rountev, and P. Sadayappan. 2008. Global Trees: A framework for linked data structures on distributed memory parallel systems. In *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '08)*. 1–13. https://doi.org/10.1109/SC.2008.5218880

[18] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 137–152. https://doi.org/10.1145/3132747.3132756

[19] Tonglin Li, Xiaobing Zhou, Kevin Br, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. 2013. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *In Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS '13)*.

[20] Chris Maynard. 2012. Comparing One-Sided Communication With MPI, UPC and SHMEM. In *Proc. Cray Users Group (CUG '12)*.

[21] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX, San Jose, CA, 103–114.

[22] MPI Forum. 2015. *MPI: A Message-Passing Interface Standard Version 3.1*. Technical Report. University of Tennessee, Knoxville.

[23] N. Namashivayam, D. Eachempati, D. Khaldi, and B. Chapman. 2015. OpenSHMEM as a Portable Communication Layer for PGAS Models: A Case Study with Coarray Fortran. In *Proc. IEEE Intl. Conf. on Cluster Computing*. 438–447. https://doi.org/10.1109/CLUSTER.2015.66

[24] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Apra. 2006. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications* 20, 2 (2006), 203–231.

[25] Qingpeng Niu, James Dinan, Sravya Tirukkovalur, Anouar Benali, Jeongnim Kim, Lubos Mitas, Lucas Wagner, and P. Sadayappan. 2016. Global-view coefficients: a data management solution for parallel quantum Monte Carlo applications. *Concurrency and Computation: Practice and Experience* 28, 13 (Jan. 2016), 3655–3671. https://doi.org/10.1002/cpe.3748

[26] OCR Working Group. 2016. The Open Community Runtime Interface, Version 1.1.0. (March 2016).

[27] OpenSHMEM Specification Committee 2016. OpenSHMEM Application Programming Interface, Version 1.3. http://www.openshmem.org. (Feb. 2016).

[28] Lisa Phillips and Brad Fitzpatrick. 2004. LiveJournal's Backend and memcached: Past, Present, and Future. In *Proc. of the 18th Conference on Systems Administration (LISA) 2004), November 14-19, 2004*, Lee Damon (Ed.). USENIX.

[29] G. Pike and J. Alakuijala. 2017. The CityHash family of hash functions (2011). http://code.google.com/p/cityhash. (Aug. 2017).

[30] Portals 4 2016. Portals 4 Reference Implementation. Online: https://github.com/Portals4/portals4. (Sept. 2016).

[31] Russell Power and Jinyang Li. 2010. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proc. 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. 293–306.

[32] John Reid. 2008. The New Features of Fortran 2008. *SIGPLAN Fortran Forum* 27, 2 (Aug. 2008), 8–21. https://doi.org/10.1145/1408643.1408645

[33] SO/IEC 1539-1:2010 2010. *Information technology – Programming languages – Fortran – Part 1: Base language*. Standard. Int'l Organization for Standardization.

[34] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*. "", 149–160. https://doi.org/10.1145/383059.383071

[35] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. 2012. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 347–353.

[36] W Scott Thornton, Nicholas Vence, and Robert Harrison. 2009. Introducing the MADNESS numerical framework for petascale computing. In *Proc. Cray Users Group (CUG '09)*.

[37] UPC Consortium. 2013. *UPC Language and Library Specifications, v1.3*. Technical Report LBNL-6623E. Lawrence Berkeley National Lab.

[38] J. J. Wilke. 2014. Coordination Languages and MPI Perturbation Theory: The FOX Tuple Space Framework for Resilience. In *Proc. IEEE Intl. Parallel Distributed Processing Symposium Workshops (IPDPSW '14)*. 1208–1217. https://doi.org/10.1109/IPDPSW.2014.136

[39] Justin M. Wozniak, Robert Latham, Sam Lang, Seung Woo Son, and Robert Ross. 2010. C-MPI: A DHT Implementation for Grid and HPC Environments. (Preprint ANL/MCS-P1746-0410). (2010).

[40] Xiaobing Zhou, Tonglin Li, Ke Wang, Dongfang Zhao, Iman Sadooghi, and Ioan Raicu. 2015. MHT: A light-weight scalable zero-hop MPI enabled distributed key-value store. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*. IEEE, 2901–2903. https://doi.org/10.1109/BigData.2015.7364116