Extending a Message Passing Runtime to Support Partitioned, Global Logical Address Spaces

D. Brian Larkins
Rhodes College

James Dinan
Intel Corporation

***Abstract*—Partitioned Global Address Space (PGAS) parallel programming models can provide an efficient mechanism for managing shared data stored across multiple nodes in a distributed memory system. However, these models are traditionally directly addressed and, for applications with loosely-structured or sparse data, determining the location of a given data element within a PGAS can incur significant overheads. Applications incur additional overhead from the network latency of lookups from remote location resolution structures. Further, for large data, caching such structures locally incurs space and coherence overheads that can limit scaling. We observe that the matching structures used by implementations of the Message Passing Interface (MPI) establish a separation between incoming data writes and the location where data will be stored. In this work, we investigate extending such structures to add a layer of indirection between incoming data reads and the location from which data will be read, effectively extending PGAS models with logical addressing.**

***Index Terms*—PGAS; Parallel Hash Table; Portals; Offload**

I. Introduction

Advances in processor architecture have led to a rise in the number of processor cores within a single compute node. While this trend continues to increase the capacity of high-performance computing (HPC) systems, the amount of memory available per core has not increased at the same rate. As a result, there has been renewed interest in both Partitioned Global Address Space (PGAS) parallel programming models as well as sparse and compact data representations.

PGAS data models are supported by a number of industry standard parallel programming frameworks, including Unified Parallel C [1], OpenSHMEM [2], Fortran 2008 Coarrays [3], and the MPI Remote Memory Access (RMA) interface [4]. Such models can be used to aggregate the memory capacity of multiple nodes in an HPC system to provide an efficient, distributed data store. However, the data stored in the PGAS is traditionally referenced using direct pointers or offsets with respect to the process memory where the data is located.

This model has been effective for storing dense, regular data (e.g., dense arrays [5]); however, it presents significant challenges to storing loosely-structured or sparse data. For such data, directly calculating the desired location in remote memory may not be possible. Instead, one or more indirections must be performed to identify the storage locations, leading to significant runtime overheads. When location resolution information is stored remotely, the application incurs additional network latency overheads for each data access. In some instances, the layout of the data may be fixed or slowly changing, allowing location resolution structures to be replicated or cached locally. Caching remote lookup structures can mitigate this impact, but consumes limited memory capacity and can cause additional overheads to maintain cache coherence.

The message matching model used by the Message Passing Interface (MPI) [4] establishes a layer of separation between a sender and the location where data is written at the receiver. Instead of specifying a direct memory location where data will be stored, the sender specifies an application-meaningful *tag* that the receiver uses to associate the transmission with a data buffer. While MPI provides a model for pushing data to a tagged location at a remote node, it does not support retrieving data from a tagged location at a remote node.

In this work, we build on networking structures designed to support MPI communication in order to establish a partitioned, global *logical* address space (PGLAS) that goes beyond traditional PGAS approaches by allowing applications to address shared, distributed data using an application-meaningful logical naming scheme rather than a direct memory address. Further, our scheme uses a scalable, low-overhead method for storing and accessing location resolution information by maintaining lookup information at the requesting node and performing location resolution during message processing.

Our work utilizes the Portals 4 network programming interface [6]. Portals provides a set of communication primitives that are intended to efficiently support a variety of HPC communication middleware systems, including MPI and PGAS models, while also lending themselves to efficient hardware implementation [7], [8].

In this paper, we describe the implementation of a parallel distributed hash table (PDHT) that supports a PGLAS data storage model and is implemented using the Portals networking interface. Our approach demonstrates that existing Portals constructs designed to support MPI messaging can be repurposed to support a PGLAS and that the novel use of these primitive communication operations can further provide a hardware-accelerated parallel hash table on systems where the Portals constructs are supported through an offload NIC.

II. Background

In this work, we explore efficient runtime support for partitioned, global logical address spaces (PGLAS). We define

COMHPC16; Salt Lake City, Utah; November 2016
978-1-5090-3829-9/16/$31.00 ©2016 IEEE

the PGLAS data model to be similar to PGAS data models, with the distinction that data elements are logically addressed using an application-meaningful *key*. Each key is mapped to a partition of the global address space and an application can query locality information for a given key or iterate over local keys, e.g., to support owner-computes models of work distribution. In contrast, PGAS models traditionally use a direct addressing model, such as a remote address (OpenSHMEM and UPC), an offset into a remote buffer (MPI), or an index into a regular array (Fortran Coarrays). While there are differences in memory addressing and layout, PGLAS utilizes the same one-sided, asynchronous data access model as existing PGAS approaches and data in global space is accessed using one-sided get (read), put (write), and atomic access functions. Thus, PGLAS is a member of the broader family of distributed key/value stores where data organization and access are similar to PGAS models.

Most PGAS models require global knowledge of data layout in order to communicate efficiently. For example, in the OpenSHMEM model, the layout of the globally accessible space must be identical at all processes. As a result, management of globally accessible memory must be performed collectively. In contrast, PGLAS introduces a layer of indirection that translates logical addresses to direct addresses. The introduction of this layer has the potential to relax the strong memory management semantics required by most PGAS models, enabling more dynamic usage models. In particular, PGLAS can support dynamic insertions and removals of key/value pairs in the global address space.

Efficient and scalable support for translation of keys to memory locations is a performance-critical, new requirement imposed by PGLAS. Traditional one-sided communication operations can satisfy the one-sided and asynchronous requirements of key translation. However, multiple operations are required to query a remote translation structure, leading to high overheads [9]. We observe that the user-supplied tag used by MPI runtimes to match an incoming message (i.e., put) with a posted receive operation supports a key/value-like remote write model. However, there are several key differences between MPI messaging and PGLAS communication. First, MPI requires message ordering, whereas PGLAS utilizes the PGAS unordered-by-default communication model. Further, when an MPI runtime receives a message for which there is no matching entry, the message is treated as unexpected and will be accepted once a matching receive operation is posted. In contrast, PGLAS models reject remote accesses to nonexistent keys. Finally, a posted receive operation is consumed by an incoming message, whereas a PGLAS model maintains the key/value association until the entry is freed.

A. Managing Distributed, Sparse Data

Of interest to the HPC community, we see PGLAS as a means for dealing with sparsity. Many common HPC applications rely on sparse data structures, which causes numerous problems in the hunt for high performance. Matrix and tensor representations often must contend with sparse data, resulting in specialized formats such as compressed sparse row (CSR) and variants. Other problems, such as N-body simulation, grid and mesh solvers, are naturally expressed using spatial decomposition trees such as quadtrees and octrees.

In these cases, there is a natural mapping between an indexing system and the corresponding data. A global distributed key/value store could replace a complicated and difficult to parallelize sparse data structure given the right application.

B. Related Work

Logically addressing data across memories in a distributed system has been explored in a number of other models. Distributed, shared key/value stores are an intrinsic part of the Piccolo programming model [10]. The Linda programming model [11] is built around tuple spaces, that allows 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 [12]. Distributed object models, such as Orca [13], reference shared data through graphs rather than direct pointers. In the Open Community Runtime (OCR) [14], 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 [4], 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++ [15], support a variety of methods for representing and querying object collections.

Hash tables are also commonly used to store sparse or loosely structured data [16], [17], [18] and their implementation has been studied in the context of MPI and popular PGAS models [19], [20], [21], [22], [23], [24] as well as in the context of RDMA offload [23], [25], [26]. In addition, PGAS-like hierarchical approaches to managing sparse spatial decomposition have been proposed [27] along with runtime methods to manage this data and exploit spatial locality [28].

III. The Portals Networking Interface

The Portals framework provides several low-level data abstractions that are used to implement PDHT, our parallel distributed hash table system. PDHT is based on a one-sided communication model to allow asynchronous data access while avoiding processor involvement at the target node. Portals provides support for both one-sided (a.k.a. non-matching) and matching communication semantics. Matching communication was designed to support MPI; we describe a novel application of the Portals matching interface abstractions to implement a one-sided model that is the basis for PDHT. By using features in the Portals library, we can take advantage of network hardware offload [7], [8] to a greater extent than if PDHT were implemented using conventional PGAS techniques. Portals defines several concepts and abstractions that are necessary to the understanding of the PDHT. The relationship between these different entities is shown in Figure 1.

Portals specifies a *network interface* (NI) as a per-process abstraction of a physical network interface on the target pro-

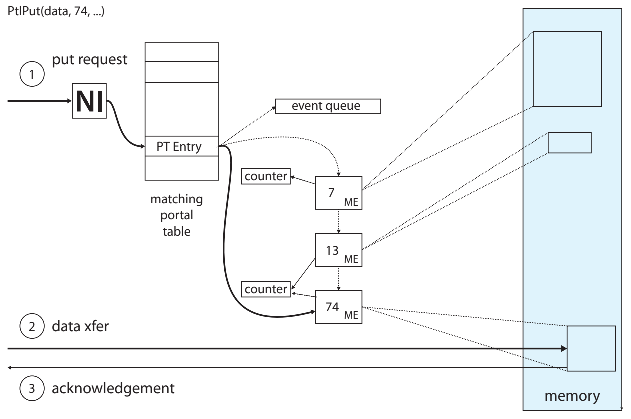

Fig. 1: Portals data structures and example put operation using the Portals matching interface.

cess. The interface is defined at creation to be either a *matching* or a *non-matching* interface, depending on the intended use for the interface (message-passing or one-sided). PDHT relies on the Portals matching interface.

Portals defines two primary roles for communication operations. The *initiator* is the process that issues Portals put, get, and atomic operation requests and the *target* is the process receiving requests. The initiator defines a *memory descriptor* (MD), which defines a region of memory to be used for Portals operations, as well as the structures used to track communication events for operations utilizing the given region. Specific completion information and notifications may be only visible to the initiator and not the target, or vice versa.

Target processing is defined in terms of a *portal table*. An index into the portal table is specified on every Portals communication operation and is used to separate and classify different channels between endpoints. Every *portal table entry* (PTE) maintains a list of memory regions that are available for communication operations as well as an event queue to track asynchronous notifications for ongoing communication.

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 all of the match criteria are satisfied, the communication operation commences working on the corresponding memory region. In particular, each ME contains a 64-bit *match bits* field that must correspond to the match bits specified on an incoming communication request. For example, a process would post an `MPI_Recv()` with a specific tag that maps to a specific match bits field. Later, Portals would receive a communication request for an `MPI_Send()` with the same match bits, which would correspond to the posted receive.

Portals provides *event queues* and *event counters* to track events relating to initiator- and target-side Portals operations. Full events contain detailed information about the event and communication operation involved, whereas counting events are lighter weight and capture only the success or failure of a given operation. Counters are useful for lightweight event counting, where copying a large amount of data from the Portals implementation to the application is unnecessary.

On the initiator, an event queue may be associated with an MD, while a queue may be associated with each PTE on the target. Counters can also be used to track both initiator and target events. Unlike target-side event queues, a counter may be associated with each ME in the match-list. Multiple MEs may share a single counter, if desired. PDHT typically uses counters to watch for successful completion and full events to track specific failures. More detailed descriptions of event and counter use within PDHT is discussed below.

To put everything together, Figure 1 shows the relationships between the target-side data structures and a sample `PtlPut()` operation. In this example, we assume that the initiator process has called `PtlPut` with some data on a matching interface, with the expected match bits set to 74. When the target process receives the request, it is dispatched through the initiator-specified PT index. Next, the match list is searched, until a 74 is found or the end of the list is reached. Once the ME has been located, the `PtlPut` data is copied into the corresponding memory region. The operation may cause a `PUT` event to be appended to the event queue or the ME success counter to be incremented. An acknowledgement is sent back to the initiator, possibly generating another full event or counting event according to the MD that the initiator supplied to the `PtlPut` operation.

IV. Implementation of PDHT

In this work, we explore the implementation of a parallel, distributed hash table (PDHT) as a first implementation of the PGLAS data model. The PDHT system implements a distributed, key/value store that is asynchronously accessible through one-sided *insert*, *read*, *write*, and *atomic update* operations. PDHT uses a hash function to map an arbitrary user-supplied key (i.e. a logical address) to a process rank and 64-bit hash value, giving each key a deterministic physical location within the distributed hash table.

A PDHT structure is treated as a distributed array of objects, with partitions spanning all processes. Upon the creation of a new distributed hash table, PDHT allocates two portal table entries (PTEs). The first PTE contains the *pending* match list and the second contains the *active* match list. The pending match list is populated with a number of entries that match any incoming communication requests (i.e. a wildcard). Each ME in the pending list corresponds to a single entry in the local portion of the distributed array. MEs in the pending list are marked as use-once and are unlinked after being written to, whereas MEs in the active list can be marked as either use-once or persistent, depending on the usage model.

The principal idea in PDHT is to create a match list entry (ME) for every element in the hash table. The 64-bit hash value produced by the hash function is used as the value for the match bits field that is provided to Portals put and get operations. By using the hash value as the match bits field, the handling of a get operation can be handled entirely by the Portals implementation.

A. *Insert Operation*

When adding a new key/value object to the hash table, the key is hashed, giving both the target processor rank as well as the match bits field to be used for the entry. In the case of a local insertion, the initiating process can copy the entry into the hash table array and locally insert a new ME with the match bits set to the hashed value.

When the entry hashes to a remote process, the initiating process performs a one-sided put operation onto the pending PTE, consuming one of the wildcard ME entries on the match list. The hash entry data is transferred and stored into the array entry that is associated with the pending entry. As shown in Figure 2 (b), at this point the table entry data resides in the correct location on the target process, but the match lists are in a transition state. To complete the operation, a new wildcard ME must replace the consumed entry on the pending list. Additionally, the active list must be updated with an ME that contains the hash. Completion of these actions results in the state shown in Figure 2 (c).

The current approach to handle the match list management between the pending and active lists is to use a dedicated progress thread. This thread blocks on the event queue associated with the pending PTE, and is automatically notified by the Portals layer when an insert request arrives. As insert requests are received, the progress thread replaces the consumed wildcard ME on the pending list and allocates a new pending list entry buffer. The thread also creates a new ME with the appropriate match bits set and appends it to the active list. Subsequent data access operations will find a match in the active list.

B. *Data Access Operations*

One-sided read, write, and atomic update operations are handled by computing the hash function on the key, then performing corresponding one-sided Portals get, put, or atomic operation, using the process rank and match bits provided by the hash. If the operation is unable to find a matching entry on the active match list, the operation reports a not-found message. If the key of the retrieved table entry doesn't match the requested key, PDHT notes the collision and either returns control to the application or performs a linear probe, depending on the desired behavior.

C. *Collisions and Key Sparsity*

Collisions occur when multiple keys map to the same rank/hash pair. In the case of a collision, PDHT utilizes a linear probing solution. The library can also be configured to simply detect collisions and defer to the aplication.

A significant difference between traditional hash tables and PDHT is that using the Portals match list for table entries detaches the link between the computed hash code and the actual location in memory. Typically, a hashed key, K, is used with modular arithmetic to determine a location in an array of size N. The hash function provides a mapping between $|K| \rightarrow N$ elements. When $|K|$ is much larger than N, the likelihood of collisions increases. In order to manage collisions, hash tables typically store a list of objects within each array entry.

PDHT also uses an array to hold the collection of hash table entries, but the indexing into this array is performed indirectly, through the ME in the match list. A PDHT hash function provides the mapping: $|K| \rightarrow P \times 2^{64}$. Each key is mapped to one of P distinct process ranks and a unique 64-bit match bits value. Compared to traditional implementations, $N \ll P \times 2^{64}$, which has the impact of dramatically reducing the likelihood of a collision, provided a reasonable hash function.

V. Experimental Evaluation

We have gathered preliminary performance results using a prototype implementation of PDHT running on the Portals reference implementation [29]. Results were gathered on the Comet 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; for our experiments, we used the Intel® Xeon® E5 nodes. There are 1944 such nodes, which are constructed from Dell⋆ PowerEdge⋆ C6320 servers containing dual 12-core Intel® Xeon® E5-2680 processors and 128 gigabytes of memory. Nodes are connected using a Mellanox⋆ FDR 56 Gb/s InfiniBand⋆ fabric.

As shown in Figure 3, we measure the weak scaling of PDHT read operations on the key/value store by examining the overall throughput of the system. Each process owns a fixed number of hash table elements which are read by a neighboring process. The system performs as expected under weak scaling, with overall system throughput increasing with the number of processes involved. Of particular interest, is the observation that throughput decreases as the number of local hash table elements increases. Lower throughputs with a larger number of elements imply higher overheads within the message matching process in the Portals runtime.

To verify this, we measure the performance of retrieving elements from the PDHT with respect to the number of elements on each process. These results are shown in Figure 4 and provide an initial characterization of the lookup cost associated with our PDHT implementation. In contrast with traditional PGAS approaches, building a remotely accessible key/value store on top of a mechanism intended to support MPI message matching can add new overheads. In particular, the Portals message processing engine must traverse the active match list until an ME that matches the given query is located. In cases where the element does not exist, the Portals layer must reach the end of the list to make this conclusion. Thus, there is a list traversal overhead that is proportional to the number of elements visited before finding a match.

The Portals layer must maintain each matching list as an ordered list in order to preserve MPI message ordering semantics. However, we observe that this ordering is not required by PDHT. Also, in contrast to MPI applications, where typical communication patterns result in short matching

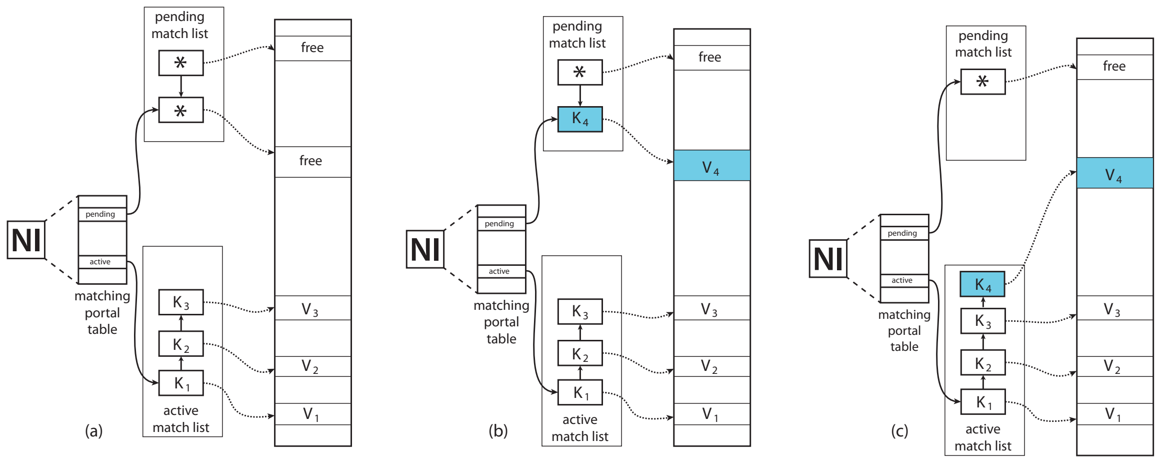

Fig. 2: Sequence of target-side actions performed during a PDHT `insert()` operation.

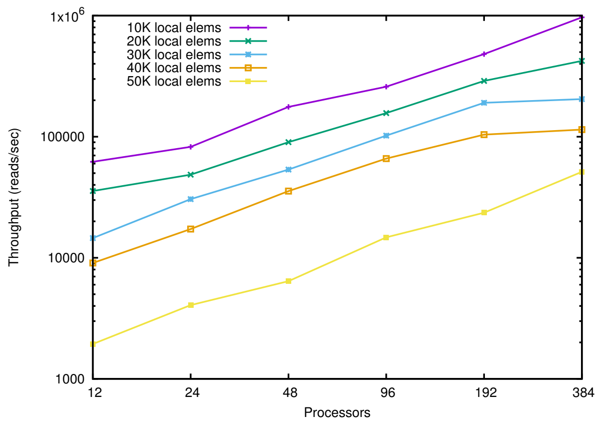

Fig. 3: System throughput for a range of local volume (entries per node).

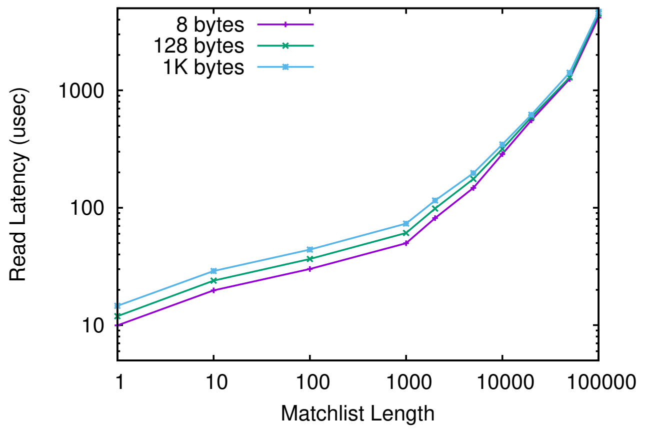

Fig. 4: Read operation latency versus list depth for various size entries.

lists or in matching occurring near the head of the list, PDHT necessarily generates long matching lists with an expected matching depth of the midpoint in the list. Thus, we identify this overhead as a key challenge to supporting such models and plan to investigate solutions in our future work.

VI. Conclusion

We have presented PDHT, a partitioned, distributed hash table implementation that uses a novel application of Portals networking interface constructs initially intended for supporting MPI messaging. By building PDHT on top of Portals, we are able to take advantage of offload NICs that support the Portals layer with a hardware processing engine. PDHT is an example of a family of partitioned, global logical address space (PGLAS) parallel programming models and we anticipate that efficient implementations of such models will be an enabling technology for sparse methods, which are of increased interest within the scientific and data analytics communities.

In this initial design study, we have identified a first mapping of the PDHT data structure to Portals primitives; however, there are a number of open areas for future work.

Currently, we use a progress thread to process incoming insertion requests. We plan to investigate the use of the proposed Portals triggered ME append operation [30] to allow PDHT to take full advantage of asynchronous progress provided through the Portals layer. However, in order for this to work, we suspect changes to Portals may be necessary, as the ME that is to be appended must also be updated with the user-supplied match bits prior to being appended to the active list.

In Section V, we identified list traversal as a significant source of overhead. A variety of solutions are possible, such as hashing within the Portals implementation, and are also of potential benefit to MPI implementations [31].

Acknowledgements

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

References

[1] UPC Consortium, "UPC language and library specifications, v1.3," Lawrence Berkeley National Lab, Tech. Rep. LBNL-6623E, Nov. 2013.

[2] "OpenSHMEM application programming interface, version 1.3," http://www.openshmem.org, Feb. 2016.

[3] J. Reid, "The new features of Fortran 2008," *SIGPLAN Fortran Forum*, vol. 27, no. 2, pp. 8–21, Aug. 2008.

[4] MPI Forum, "MPI: A message-passing interface standard version 3.1," University of Tennessee, Knoxville, Tech. Rep., June 2015.

[5] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, applications and performance of the global arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006.

[6] B. W. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson, "The Portals 4.0.2 network programming interface," Sandia National Laboratories, Tech. Rep. SAND2013-3181, April 2013.

[7] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson, "SeaStar interconnect: Balanced bandwidth for scalable performance," *IEEE Micro*, vol. 26, no. 3, pp. 41–57, May 2006.

[8] S. Derradji, T. Palfer-Sollier, J. P. Panziera, A. Poudes, and F. W. Atos, "The BXI interconnect architecture," in *Proc. IEEE 23rd Annual Symposium on High-Performance Interconnects*, Aug 2015, pp. 18–25.

[9] N. Namashivayam, D. Eachempati, D. Khaldi, and B. Chapman, "OpenSHMEM as a portable communication layer for PGAS models: A case study with Coarray Fortran," in *Proc. IEEE Intl. Conf. on Cluster Computing*, Sept 2015, pp. 438–447.

[10] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 293–306.

[11] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *IEEE Computer*, vol. 19, no. 8, pp. 26–34, Aug. 1986.

[12] J. J. Wilke, "Coordination languages and MPI perturbation theory: The FOX tuple space framework for resilience," in *Proc. IEEE Intl. Parallel Distributed Processing Symposium Workshops*, ser. IPDPSW '14, May 2014, pp. 1208–1217.

[13] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A Language For Parallel Programming of Distributed Systems," *IEEE Trans. on Software Eng.*, vol. 18, no. 3, pp. 190–205, Mar. 1992.

[14] OCR Working Group, "The open community runtime interface, version 1.1.0," March 2016.

[15] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proc. Conf. on Object Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93, 1993, pp. 91–108.

[16] L. Phillips and B. Fitzpatrick, "Livejournal's backend and memcached: Past, present, and future," in *Proceedings of the 18th Conference on Systems Administration (LISA) 2004), Atlanta, USA, November 14-19, 2004*, L. Damon, Ed. USENIX, 2004.

[17] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM*, 2001, pp. 149–160.

[18] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: an interaction and coordination framework forcoupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[19] T. Li, X. Zhou, K. Br, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "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*, ser. IPDPS '13, 2013.

[20] R. Gerstenberger, M. Besta, and T. Hoefler, "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, 11 2013, pp. 53:1–53:12.

[21] J. M. Wozniak, R. Latham, S. Lang, S. W. Son, and R. Ross, "C-MPI: A DHT implementation for grid and HPC environments. (Preprint ANL/MCS-P1746-0410)," 2010.

[22] C. Maynard, "Comparing one-sided communication with MPI, UPC and SHMEM," in *Proc. Cray Users Group*, ser. CUG '12, 2012.

[23] P. Stuedi, A. Trivedi, and B. Metzler, "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*, G. Heiser and W. C. Hsieh, Eds. USENIX Association, 2012, pp. 347–353.

[24] X. Zhou, T. Li, K. Wang, D. Zhao, I. Sadooghi, and I. Raicu, "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, 2015, pp. 2901–2903.

[25] C. Mitchell, Y. Geng, and J. Li, "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)*. San Jose, CA: USENIX, 2013, pp. 103–114.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 295–306.

[27] D. B. Larkins, J. Dinan, S. Krishnamoorthy, S. Parthasarathy, A. Rountev, and P. Sadayappan, "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*, ser. SC '08, Nov 2008, pp. 1–13.

[28] D. B. Larkins, "Improving data locality for irregular partitioned global address space parallel programs," in *Proc. 50th Annual Southeast Regional Conference*, ser. ACM-SE '12. New York, NY, USA: ACM, 2012, pp. 280–285.

[29] "Portals 4 reference implementation," Online: https://github.com/Portals4/portals4, Sep. 2016.

[30] T. Schneider, T. Hoefler, R. Grant, B. Barrett, and R. Brightwell, "Protocols for Fully Offloaded Collective Operations on Accelerated Network Adapters," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct. 2013, pp. 593–602.

[31] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI message matching misery," in *Proc. 31st Intl. Conf. High Performance Computing*, ser. ISC '16, 2016, pp. 281–299.