2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

SHAD: the Scalable High-performance Algorithms and Data-structures Library

Vito Giovanni Castellana, Marco Minutoli
Pacific Northwest National Laboratory
Richland, WA, USA
{vitoGiovanni.castellana, marco.minutoli}@pnnl.gov

***Abstract*—The unprecedented amount of data that needs to be processed in emerging data analytics applications poses novel challenges to industry and academia. Scalability and high performance become more than a desirable feature because, due to the scale and the nature of the problems, they draw the line between what is achievable and what is unfeasible. In this paper, we propose SHAD[1], the Scalable High-performance Algorithms and Data-structures library [1].**

SHAD adopts a modular design that confines low level details and promotes reuse. SHAD's core is built on an Abstract Runtime Interface which enhances portability and identifies the minimal set of features of the underlying system required by the framework. The core library includes common data-structures such as: Array, Vector, Map and Set. These are designed to accommodate significant amount of data which can be accessed in massively parallel environments, and used as building blocks for SHAD extensions, i.e. higher level software libraries.

We have validated and evaluated our design with a performance and scalability study of the core components of the library. We have validated the design flexibility by proposing a Graph Library as an example of SHAD extension, which implements two different graph data-structures; we evaluate their performance with a set of graph applications. Experimental results show that the approach is promising in terms of both performance and scalability. On a distributed system with 320 cores, SHAD Arrays are able to sustain a throughput of 65 billion operations per second, while SHAD Maps sustain 1 billion of operations per second. Algorithms implemented using the Graph Library exhibit performance and scalability comparable to a custom solution, but with smaller development effort.

***Index Terms*—Data structures, Distributed systems, Parallel programming, Software library, Big data, High performance computing**

I. Introduction

Emerging data analytics methodologies aims at addressing the 5 Vs of Big Data [2]: Volume, Velocity, Variety, Variability and Value. Processing an unprecedented volume of heterogeneous data that is rapidly changing poses novel challenges to industry and the research community. Under these working conditions, providing High-Performance and scalable solutions becomes a fundamental requirement. Intuitively, when the data does not fit in the memory of a conventional server, the application should be able to scale in size and performance on multi-node systems. Due to the space and computational complexity of today's analytics workloads, scalability and high-performance not only imply a *faster* time to solution but also may push forward the limit between what is achievable and what is unfeasible in human time.

Conventional approaches tackle the Big Data challenges highly customizing the applications to target specific machines or architectures. Alternatively, another common and more aggressive solution is based on hardware-software co-design which customizes both the software implementation and the actual target hardware to solve a specific class of problems [3]. These approaches are characterized by significant design and development effort, long time-to-solution, and low portability and flexibility. To address these issues, we propose SHAD, the Scalable High Performance Algorithms and Data-structures library. SHAD is designed to achieve flexibility and portability, while providing scalability and high performance. Unlike other high performance data analytics frameworks [4], [5], SHAD can support different application domain, including, but not limited to, graph processing, machine learning, and data mining.

The SHAD software stack is composed of three layers: an Abstract Runtime Interface, a collection of general purpose algorithms and data-structures, and domain specific libraries called *SHAD extensions*. The Abstract Runtime Interface exposes a set of primitives for managing tasks execution and concurrency, and data-movements on both clusters and single-node machines, with the purpose of hiding low level details of the underlying runtime systems and architectures. Using the Abstract Runtime Interface, we define the core SHAD data-structures layer, which includes general purpose containers such as sets, maps, and vectors. Such data-structures expose interfaces inspired by the C++ Standard Template Library, improving developers productivity and possibly facilitating SHAD adoption in already existing code bases. The core SHAD components offer enough features to implement a wide variety of applications, and they represent the building blocks for SHAD extensions, such as linear algebra and graph libraries. Extensions can be developed as layered libraries, and they can use multiple lower-level libraries. This unique feature makes SHAD not only a programming library, but an evolving framework supporting a variety of big data analytics and high performance computing applications. SHAD adopts a shared-memory programming model, even though data can be actually spread over different nodes of a cluster. To achieve high-performance, SHAD makes extensive use of asynchronous (non-blocking) parallel execution, and active messages. In

[1]Source code available on GitHub at https://github.com/pnnl/SHAD

978-1-5386-5815-4/18/$31.00 ©2018 IEEE
DOI 10.1109/CCGRID.2018.00071

particular, the data-structures access methods allow to automatically migrate the computation where the data are located, rather than gathering them. This is particularly beneficial when the same operations (e.g., a function) is applied to data. A typical example is visiting all the nodes/edges of a graph, or updating all the elements of an array.

In this paper we describe the design of SHAD, detailing features and adopted solutions for its components. The main contributions of our work are:

- the definition of an Abstract Runtime Interface, used by the upper layers of the SHAD software stack, that identifies the minimal features set of the underlying systems required by SHAD;
- the design of a general data-structure template, which can be adopted to implement data-structures on distributed systems;
- the design and implementation of a set of general purpose data-structures, using the generic template; such data-structures can be composed to obtain more complex data-structures and libraries, called SHAD extensions;
- two variants of an example SHAD extension of a graph library: the first is based on the Compressed Sparse Row (CSR) representation while the second is based on indexing;
- a performance and scalability evaluation of the general purpose data-structures and of example applications implemented using the graph libraries.

The remainder of the paper proceeds as follows. Section II overviews the proposed methodology and design; more in detail: Section II-A describes the rationale behind the Abstract Runtime Interface, and its main characteristics; Section II-B illustrates the general layout designed for the SHAD data-structures; Section II-C describes the core, general purpose SHAD data-structures. Section III introduces SHAD extensions, and describes graph libraries as practical examples. Section IV validates the proposed approach through experimental performance and scalability studies. Section V overviews related work, and Section VI concludes the paper, and anticipates future work.

II. SHAD Design

One of the main objective of SHAD is to provide a simple, productive, programming model. For this purpose, SHAD adopts a shared-memory abstraction at the user level, even though actual data can be distributed on several nodes of a cluster and operations can be executed remotely and in parallel. SHAD is designed as a software stack, composed of three layers: the Abstract Runtime Interface, the SHAD core library which includes general purpose data-structures, and, SHAD extensions, which are domain specific libraries obtained by composing SHAD data-structures and possibly other extensions.

A. The Abstract Runtime Interface

The Abstract Runtime Interface is designed to achieve portability of SHAD on different platforms, by decoupling the upper layers of the stack from the underlying architecture. We have firstly explored the adoption of a runtime abstraction layer with similar objectives in [6]. From our previous effort, we preserve the concept of *machine abstraction*, which models the hardware platform as a set of *localities*. We define a locality as a unit of computation with directly accessible memory: depending on the characteristics of the underlying system, localities can model cores, NUMA domains, or nodes in a distributed system. Similar concepts are adopted for example in [7].

In SHAD, we extensively use remote and asynchronous execution through active messages [8]. Active messages are used to reduce expensive data movements across localities, and thus reduce network traffic. For example, several data-structures access methods, provided in the upper layers of the stack, exploit active messaging to automatically move the computation where the data are located, rather than gathering them locally. This is particularly beneficial when the same operations (e.g., a function) is applied to big portion of the data-structures. A typical example is visiting all the nodes/edges of a graph or updating all the elements of an array. Asynchronous execution is adopted to exploit parallelism, and, in case of asynchronous remote execution, also to tolerate the communication latency of the network. In fact, it has been shown that asynchronous execution is beneficial in several applications areas, for example it can help to speed up convergence for linear systems [9], speed up belief propagation [10], and stochastic optimization [11].

The Abstract Runtime Interface supports asynchronous execution by associating identifiers, called *handles*, to spawned tasks; multiple tasks can be associated to the same handle. The runtime interface includes wait commands, which allow checking for the termination of the asynchronous operations associated with a given handle. The main methods offered by the Abstract Runtime Interface are:

- **[async]ExecuteAt** [asynchronously] execute a function on a given locality;
- **[async]ExecuteAtWithRet** [asynchronously] execute a function with a return value on a given locality;
- **[async]ExecuteOnAll** [asynchronously] execute a function on all localities;
- **[async]ForEachAt** [asynchronously] execute a parallel loop on a given locality;
- **[async]ForEachOnAll** [asynchronously] execute a parallel loop on all localities;
- **waitForCompletion** wait for the completion of asynchronous tasks.

Listing 1 shows an example program written using the Abstract Runtime Interface: when working at this level of abstraction, users need to explicitly manage data locality. One of the purposes of the higher-level APIs is to hide such complexity. In this work, we use the Global Memory and Threading (GMT) runtime system [12] as backend for the SHAD Abstract Runtime Interface implementation. Specifically, we only employ the software multi-threading, the active

```
// each locality has its own counter variable
std::atomic<size_t> counter(0);

int main(int argc, char ** argv) {
  auto incrLambda = [] (shad::rt::Handle &handle,
                        size_t &incr) {
    // do something
    counter += incr;
  };
  size_t args = 1;
  shad::rt::Handle handle;
  // execute the specified function on all localities
  shad::rt::asyncExecuteOnAll(handle, incrLambda, args);
  // wait for termination of asynchronous tasks
  shad::rt::waitForCompletion(handle);

  // synchronous reduce
  size_t overallCnt;
  auto getLambda [] (size_t& args, size_t* retValue) {
    // do something
    *retValue = counter;
  };
  for (auto loc : shad::rt::allLocalities()) {
    size_t remoteCnt = 0;
    // synchronous remote execution, with return value
    shad::rt::executeAtWithRet(loc, getLambda,
                               args, &remoteCnt);
    overallCnt += remoteCnt;
  }
  ...
  return 0;
}
```

Listing 1. Example program using SHAD Runtime Interface Methods.

messaging, and the message aggregation modules of GMT. The Abstract Runtime Interface provides an additional layer of abstraction that simplifies the porting of SHAD to different runtime systems and computer architectures by only imposing a small set of requirements. This enabled us to use sub-components of a much more complex runtime system, as this implementation demonstrates.

B. Data-structures Layout

All the SHAD data structures are implemented as distributed global objects, which provide the user with a shared memory abstraction on distributed memory machines. In this work, we define a general data-structure template, which can be adopted as a design pattern for data-structures on distributed systems. Figure 1 shows the main components of such a template. Each locality has a portion of the data (*local* data), a *catalog* for the data-structure, and an aggregation buffer. All the interaction between these components is obtained through the data-structure interface, which also serves as an API for external users.

Local Data: The proposed design pattern does not make any specific assumption on the kind of storage used for the local data. In our current implementation of SHAD, all the data-structures are stored in main memory to provide high performance. Nevertheless, it is also possible to adopt other solutions. For example, a promising approach that we plan to explore in future works, is to exploit persistent storage (e.g., SSDs) to support mechanism for reliability.

Catalog: The object catalog, which is replicated on each locality, manages the objects' life cycle. When a distributed object is created, the catalog associates a unique identifier to

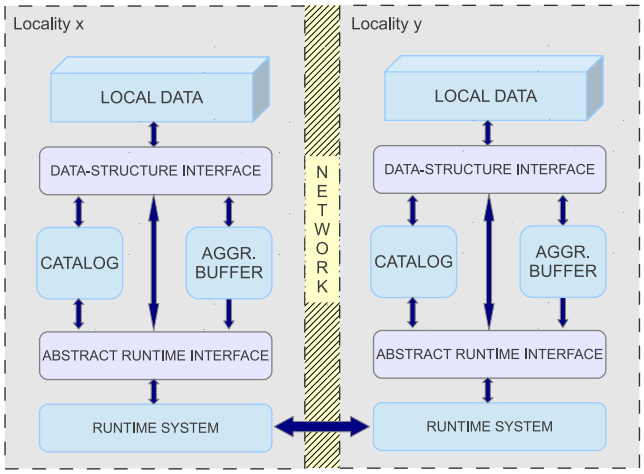

Figure 1. Generic SHAD data-structures design.

the newly created instance. Once the catalog has assigned an ID to the new object, it sends an active message to all the other localities in the system to allocate the local portion of the data structure. To avoid the complex synchronization across the system for assigning identifiers to new objects, the space of the valid object identifiers is partitioned between the available localities. Consequently, each locality has a limited but very large number of object that can be concurrently instantiated for each data-structure type (2^{48} in the current implementation). The assigned object identifier will act as a global pointer to the instantiated global object across the system so that, when the computation moves from one locality to another, it can be used to access the local portion of data of the distributed object.

Aggregation Buffers: Given the shared-memory abstraction, populating a data-structure may require transferring data to a remote locality. Performing many fine-grained data transfers across the network of a distributed system is typically not efficient: for this reason, our design template includes aggregation buffers for data movements. To improve performance, we allocate on each locality one aggregation buffer per remote destination, so they can be managed independently. The size of the aggregation buffers can be tuned to better suit the characteristics of different networks and/or runtime systems.

Data Structure Interface: The data structure interface defines the behavior of the data structure and provides a global, shared memory view of the associated objects. All SHAD data-structure interfaces offer a common subset of operations, which have similar behaviors. These are:

- **Create** create a new data-structure, allocating local data (on all localities) and assigning a new identifier;
- **Destroy** destroy the data-structure, invalidating the associated identifier and freeing (on all localities) the local data memory;
- **[async]Insert** [asynchronously] insert an element in the data-structure;
- **[async]Delete** [asynchronously] remove a specific ele-

ment from the data-structure; currently, only hash-based data-structures support delete operations;
- **[async]Lookup** [asynchronously] retrieve a specific element from the data-structure;
- **[async]Apply** [asynchronously] apply a function to a specific element, eventually, with additional user-provided arguments;
- **[async]ForEachElement** [asynchronously] apply a function to all the elements in the data-structure, eventually, with additional user-provided arguments.

Intuitively, each data-structure may provide additional data-structure specific methods, algorithms, and variations of the above mentioned ones. The underlying implementations use the catalog, the aggregation buffer, and, the services provided by the abstract runtime interface, to move computation and data between different localities.

Implementation Details: In the current implementation of SHAD, an *Abstract Data Structure* (ADS) class serves as base class of all the SHAD data-structures and implement the previously described template. The ADS provides all the common functionalities, such as, for example, the catalog management. This software solution aims at facilitating the development of additional data-structures, and at providing a more consistent interface between them.

C. SHAD General Purpose Data-structures

The SHAD core library is a collection of commonly used, general purpose data structures, which currently includes Array, Vector, Map and Set. All the core data structures are designed to be thread safe, so they can be accessed concurrently from and within any locality. They expose high-level APIs inspired by popular programming languages libraries, and in particular, by the C++ Standard Template Library (STL). As in the STL, all data-structures are templated, so they can store arbitrary data. The adopted APIs have the purpose of:

- hiding all the low-level details of the underlying architecture;
- increasing the user-productivity without sacrificing scalability and performance;
- facilitating the adoption of SHAD in existing codebases, thanks to their similarity to the well known STL interfaces.

Nevertheless, SHAD APIs also provide (mainly through overloading) advanced users with hooks for customizing the data layouts or access methods behavior: for example, it is possible to customize the data distribution, and the behavior of insert/update operations, for which the default is overwriting existing data, if any. In the following we detail the key features of the SHAD core data-structures:

a) ***Array:*** Conforming with the C++ STL array, the SHAD Array has fixed size, which is defined when the object is created. In our current implementation data are spread evenly across the system by default in order to maximize data locality when accessing contiguous elements. Nevertheless, the experienced user could specify alternative data distribution policies.

```
shad::Map<size_t, size_t>::ObjectID CreateAndPopulate(
     size_t expSize) {
  // create a new Map object
  auto mapPtr = shad::Map<size_t, size_t>::Create(expSize);
  shad::rt::Handle handle;
  for (size_t i = 0; i < expSize; ++i) {
    mapPtr->AsyncInsert(handle, i*2, i+1);
  }
  shad::rt::waitForCompletion(handle);
  // return the Global Identifier of the Map
  return mapPtr->GetGlobalID();
}
```

Listing 2. Usage example of SHAD Map Data-structure.

b) ***Vector:*** SHAD vectors are basically dynamically expandable arrays. Efficient dynamic expansion is achieved by striping the data across the system in fixed size chunks. This data layout and the dynamic expansion of the container allows supporting push-back operations and insert of data-sequences larger than the current size of the container. For performance reasons, SHAD vectors currently do not provide STL-like insert operations, which allow inserting a new element in any position of the sequence. In fact, the standard behavior of the STL insert functions prescribes that all the data beyond the insertion point must be moved, in order to accommodate the newly inserted data. While this is feasible when managing small amount of data on a single node system, it would result in severe performance penalties when working with big data, especially on distributed system, which indeed represent the main target of the proposed library. Updates are not affected by these issues, and can be performed on any position of the vectors.

c) ***Map:*** SHAD maps are distributed unordered hash-maps that support concurrent insertion, update and deletion. Supporting both concurrent insertion and deletions is particularly desirable for streaming applications. Maps local data is organized in buckets which can grow dynamically, implemented as thread-safe linked lists. We achieve safety through a relatively simple, but effective, locking scheme. In case of insertion/deletion at a given position in a bucket, only such position and the subsequent ones are locked, while all the previous ones can be accessed. Update operations instead do not prevent operations on any other positions. For space efficiency, delete operations replace the element to be deleted with the last element in the bucket, and free the slot of the moved element. The keys space is partitioned using hashing at two different levels: the first is needed to uniquely identify the locality associated with a key, and the second level identifies the associated bucket in the local storage. We currently use, for both levels, the *one-at-a-time Jenkins* hash functions [13]. Listing 2 shows a code snippet in which a new Map instance is created and populated.

d) ***Set:*** SHAD sets provide distributed unordered sets, with same data layout and features of the map.

III. SHAD Extensions

SHAD *extensions* are high-level libraries built on top of the lower layers of the SHAD stack, and/or by combining

other extensions. In this work, we validate this design idea by providing an example extension of a Graph Library. In order to demonstrate SHAD flexibility, we implement two different graph data-structures: a plain Compressed Sparse Row (*CSR*) graph, and an index-based (*Edge Index*) representation. The CSR implementation solely uses SHAD arrays, while the latter combines SHAD maps and sets.

Both graph data-structures expose the same user-level interface; the core visit operations offered in the API are:

- **[async]ForEachEdge** [asynchronously] execute a function for all the edges in the graph;
- **[async]ForEachVertex** [asynchronously] execute a function for all the vertices in the graph;
- **[async]ForEachIncidentEdge** [asynchronously] execute a function for all the incident edges of a vertex in the graph;
- **[async]ForEachNeighbor** [asynchronously] execute a function for all the neighbors of a vertex in the graph;

CSR Graph*:* The *CSR Graph* represents the graph as an adjacency matrix using a compressed sparse row format. A CSR based graph representation is constituted by two arrays $\bar{v}$ and $\bar{e}$). Given a graph $G = (V, E)$, the array $\bar{e}$ has length $|E|$ and stores contiguously the neighbor lists of all the vertices $v \in V$. The array $\bar{v}$ has length $|V| + 1$ and, for each vertex $v \in V$, stores the starting and ending point of its neighbor list in the $\bar{e}$ array.

Edge Index Graph*:* Indexes can be used to conveniently represent graphs. Our index-based graph implementation uses a map of sets (storing the neighbor lists) as its underlying storage. Given a graph $G = (V, E)$, the map stores key/value pairs of the form $(v_i, \delta(v_i))$, where $v_i \in V$ and $\delta(v_i) = \{v : (v_i, v) \in E\}$. As aforementioned, the default behavior of insert/update methods is to overwrite existing data, if any; nevertheless, to support this representation, we use an *append* insert/update policy, so that an insert operation in the map would actually perform an insert in the storage used as value (i.e., the set). This approach can be used in general to implement multi-value associative containers.

The only difference, in terms of features, between the two different implementations, is that the CSR Graph is a *static* data-structure, while the Edge Index Graph is *dynamic*. This means that our Edge Index Graph can dynamically grow and shrink, and its API offers edge insert and delete methods. This feature allows developing applications on streaming data. It is important to notice that this limitation of the CSR Graph is not due to our specific implementation, but to the CSR representation itself. In fact, even if in principle it is possible to update a CSR graph, it is typically not feasible in practice, since in the worst case insert and delete operations would require to update the whole data-structure (e.g., for recomputing the offsets).

Graph Applications*:* We validate the effectiveness of the SHAD design by developing two different applications on top of the Graph extension: PageRank and Triangle Counting (shown in Algorithm 1 and 2, respectively). We selected them mainly because of their relevance (they represent critical kernels in several more complex big data applications) and because they are representatives of two different classes of graph algorithms. Triangle Counting is indeed representative of graph exploration and pattern matching algorithms, while PageRank is a representative of *vertex-centric* algorithms, operating mostly on vertices and their neighbor lists. They also exhibit very different behaviors, especially in our current implementation: Triangle Counting is completely data flow driven and asynchronous (each triangle is matched independently through parallel graph traversals), while PageRank is phase synchronous (and thus potentially affected by high synchronization costs).

We highlight that in this work we have purposely not adopted any advanced algorithmic optimization or heavy data-structure customization, with the objective of demonstrating that SHAD allows to quickly implement both extensions and applications, without necessarily renounce performance and scalability. Listing 3 shows an example of how the three nested loop of Algorithm 1 can be implemented with SHAD using the CSR Graph. The Edge Index implementation differs only in the type of the graph object.

```
Input: A graph G = (V, E)
Output: Number of unique triangles in G
Triangles = 0;
forall edges (i, j) ∈ E s.t. i < j do asynchronously
    forall k ∈ δ(j) s.t. j < k do asynchronously
        forall w ∈ δ(k) s.t. k < w do asynchronously
            if w == i then
                Triangles++;
            end
        end
    end
end
return Triangles;
```

Algorithm 1: Triangle Counting Algorithm.

IV. Experimental Results

In this section we validate our design and evaluate the scalability and performance of our proposed SHAD framework. All the experiments have been conducted on a cluster of 24 nodes, equipped with two Intel Xeon E5-2680 v2 CPUs working at 2.8 GHz (hyper-threading disabled) and 768GB of memory per node. In all the experiments, we have abstract the system using a locality for each socket of the allocated nodes in order to improve intra-locality performance. We scale our experiments up to 320 cores, distributed on 32 SHAD localities. We start our analysis by evaluating the performance of the SHAD core data-structures. For conciseness, we report results of only a subset of them: Array, and Map. The choice is motivated by the fact that these are the main building blocks for our Graph Library extensions and also that Vectors and Sets exhibit very similar behavior when compared to Arrays and Maps respectively.

Input: A graph $G = (V, E)$
Output: Ranks for all the $v \in V$
dump = 0.85;
baseRank = (1 - dump) / $|V|$;
forall *vertices* $v \in V$ **do** asynchronously
  rank[v] = $1/|V|$;
end
$error$ = 0;
repeat
  $error$ = 0;
  in[$|V|$];
  out[$|V|$];
  forall *vertices* $v \in V$ **do** asynchronously
    out[v] = rank[v] / $|\delta(v)|$;
    forall $j \in \delta(v)$ **do** asynchrnous
      in[v] += out[j];
    end
    oldRank = rank[v];
    rank[v] = baseRank + dump * in[v];
    $error$ = |rank[v] - oldRank|;
  end
until $error < \epsilon$;
return rank;

Algorithm 2: PageRank algorithm.

```
1  shad::rt::Handle handle;
2
3  using GraphType = shad::CSRGraph<size_t>;
4  // ... or using GraphType = shad::EdgeIndexGraph<size_t>;
5
6  // G is the Object Identifier of the Graph
7  auto GraphPtr = GraphType::GetPtr(G);
8  GraphPtr->AsyncForEachEdge(
9    handle,
10   [](shad::rt::Handle & handle,
11      const size_t & i, const size_t & j,
12      GraphType::ObjectID & G) {
13     if (i >= j) return;
14     auto GraphPtr = GraphType::GetPtr(G);
15     GraphPtr->AsyncForEachNeighbor(
16       handle, j,
17       [](shad::rt::Handle & handle,
18          const size_t & j, const size_t & k,
19          GraphType::ObjectID & G,
20          const size_t & i) {
21         if (j >= k) return;
22         auto GraphPtr = GraphType::GetPtr(G);
23         GraphPtr->AsyncForEachNeighbor(
24           handle, i,
25           [](shad::rt::Handle & handle,
26              const size_t & i,
27              const size_t & w, const size_t & k) {
28             if (w != k) return;
29
30             TriangleCounter++;  // atomic counter.
31           }, k);
32       }, G, i);
33   }, G);
34 shad::rt::waitForCompletion(handle);
```

Listing 3. Triangle Counting example with SHAD.

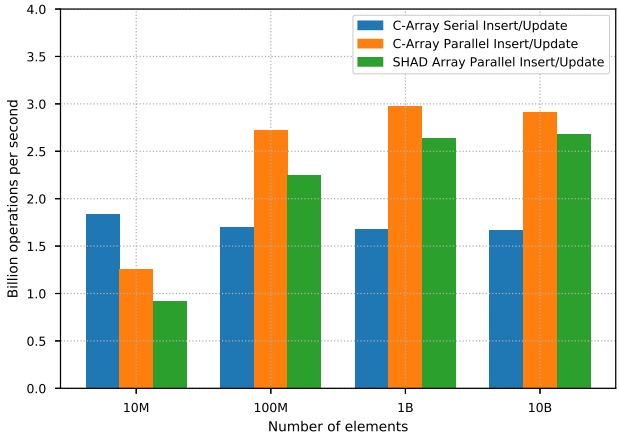

Figure 2. Insert/Update performance of SHAD Arrays compared to C/C++ plain arrays of integers.

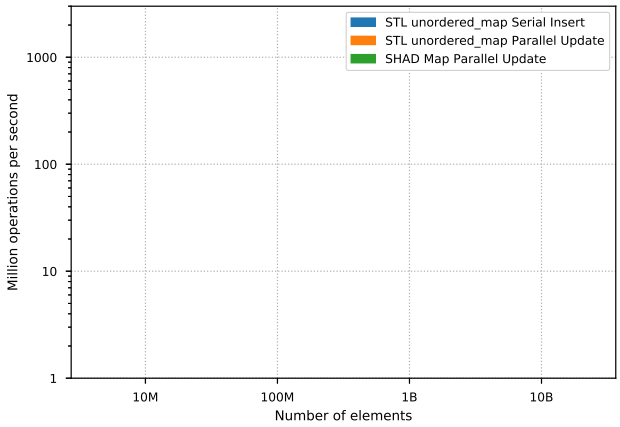

Figure 3. Insert/Update performance of SHAD Maps compared to STL unordered maps of ⟨integer, integer⟩ pairs.

Performance on a Single Locality: All SHAD data-structures are designed to be thread-safe, allowing concurrent access at both inter and intra-locality levels. In order to quantitatively measure the overhead introduced to achieve thread-safety, we compare insert/update performance of SHAD data-structures against their non-thread safe counterparts.

Figure 2 compares performance of SHAD Arrays against C/C++ plain arrays, varying the size of the data-structures from 10 Millions to 10 Billions elements (integers). For each data size we perform 3 experiments: serial insertion in the C array, parallel insertion in the C array (using the *ForEach* method of SHAD's abstract runtime interface) and parallel insertion in the SHAD array. For the serial update, as expected, we do not see any substantial throughput variation, even though experiments show a slight degradation for bigger sizes. When looking at parallel execution instead, we observe, for both the C and SHAD arrays, significant throughput increase especially when the size of the container grows from 10 to 100 Millions

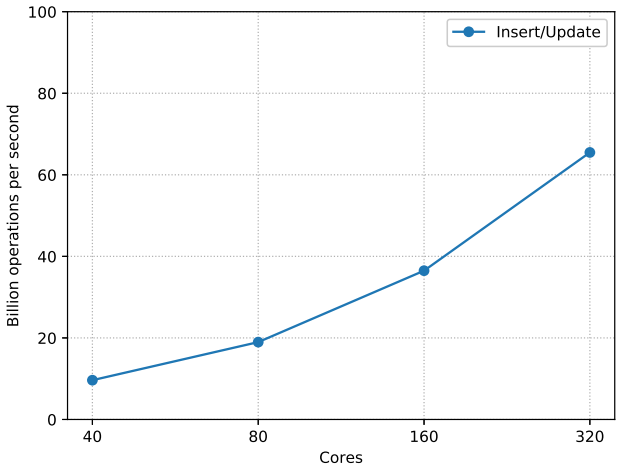

Figure 4. Strong scaling analysis of the SHAD Array when inserting 10 Billion elements (integers).

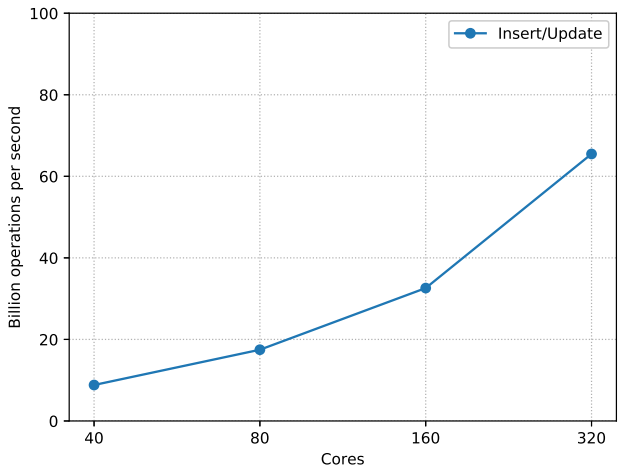

Figure 5. Weak scaling analysis of the SHAD Array when inserting 625 Milion elements per node with the last point inserting 10 Billion integers overall.

elements. When comparing the two data-structures at bigger sizes, we observe similar performance: this demonstrates that SHAD arrays introduce very limited overhead when working on a single locality.

Figure 3 compares performance of SHAD maps against STL unordered maps. In this case, when evaluating parallel accesses, we only consider values updates: STL maps are indeed not thread safe, and thus, concurrent insertions of key-value pairs can lead to data-races and undefined behaviors. As for the previous experiments, we observe the biggest throughput increase when increasing the size from 10 to 100 Millions elements, after which, we report smaller performance variations. As for the arrays, SHAD maps and STL unordered maps exhibit comparable performance.

Performance on Multiple Localities: We have analyzed the performance and the scalability of the SHAD Array and Map varying the size of the input data from 10 Millions to 10 Billions elements. Figure 4 and Figure 5 illustrate performance and scalability of the SHAD Array. Experimental results show that the Array scales almost linearly for both weak and strong scaling, with a maximum throughput of 65.5 Billion operations per second. Figure 6 and Figure 7 show results for the performance and scalability of the SHAD Map, for Insert and Update operations.

The experiments show that Insert operations are considerably more expensive than the Updates. The difference in performance between the two operations is caused by the additional cost of the memory allocation for the buckets, and, of the locking scheme during insertions. In fact, as detailed in Section II-C, insert operations lock all the subsequent positions of the destination bucket with the consequence that, in the case of collisions, multiple insertions in the same bucket may be serialized. Nevertheless, our measures show that both operation have almost linear scalability until we reach saturation (for the updates) at 1 Billion of operations per second.

Graph Applications: Figure 8 illustrates performance and scalability (strong scaling) of our PageRank implementation when varying the number of cores. For these experiments, we use a scale-20 Erdös-Rényi graph (1048576 vertices, 21808992 edges) from [14]. Results show overall good scalability when using both the CSR Graph and the Edge Index graph implementations, especially with higher core-counts. Nevertheless, when transitioning from 40 to 80 cores, we see relatively lower performance gains for the Edge Index Graph, and even performance degradation for the CSR Graph. This is due to the increasing pressure on the network, which is not well compensated by the increased number of computing resources. The Edge Index Graph is less affected by this issue, because in our current implementations the neighbors list of a vertex is co-located with it, thus improving locality.

Figure 9 and Figure 10 illustrate performance and scalability (strong scaling) of the Triangle Counting applications. In addition to our SHAD implementation, we also show results obtained with a custom, low-level and optimized implementation of the algorithm, developed using the complete GMT runtime library, including its Partitioned Global Address Space (PGAS) support. For these experiments, we use a scale-23 Erdös-Rényi graph (8388608 vertices, 200622558 edges). As shown in Figure 9, the CSR Graph implementation provides overall better scalability than the Edge Index counterpart, which in this case, suffers performance degradation when shifting from 40 to 80 cores (but still providing good scalability with higher core counts). Nevertheless, the Edge Index implementation provides much better performance, thanks to its better data-locality.

Figure 10 details the performance characteristics of the SHAD (Edge-Index) implementation and the GMT implementation only, facilitating their comparison. The custom GMT implementation, as expected, is faster than the high-

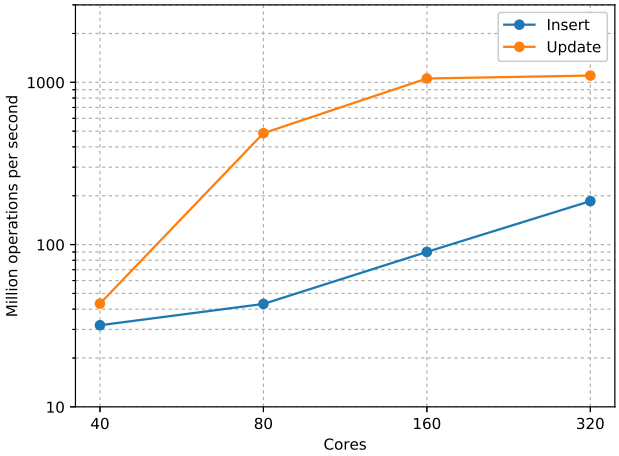

Figure 6. Strong scaling analysis of the SHAD Map when inserting and updating 10 Billion ⟨integer, integer⟩ pairs. Results are plotted in log scale.

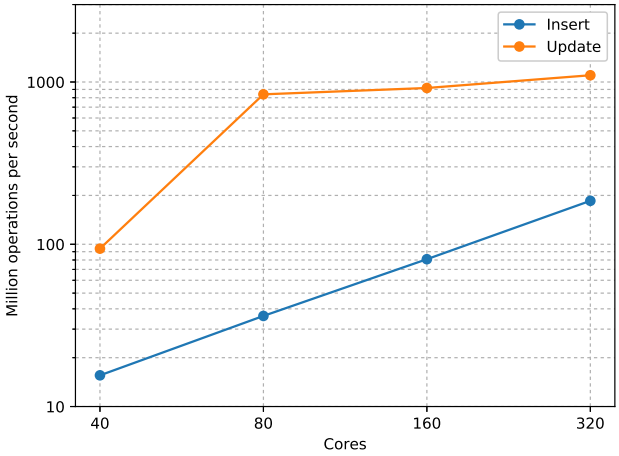

Figure 7. Weak scaling analysis of the SHAD Map when inserting and updating 625 Million ⟨integer, integer⟩ pairs per node with the last point inserting 10 Billion pairs overall. Results are plotted in log scale.

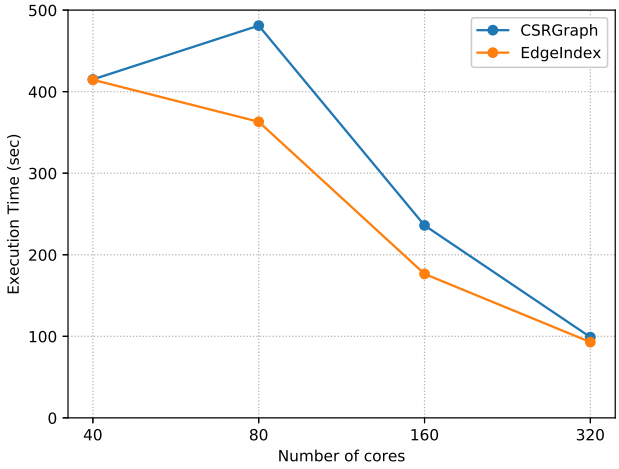

Figure 8. Strong scaling of PageRank using the Edge Index and the CSRGraph to store an Erdös-Rényi Graphs with scale factor 20.

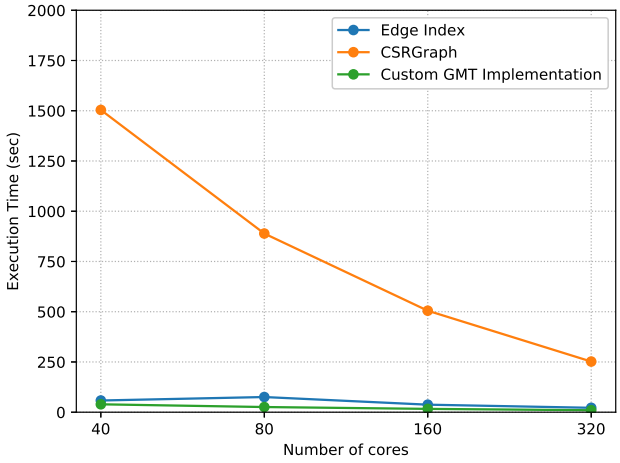

Figure 9. Strong scaling of the Triangle Counting on a Erdös-Rényi Graphs with scale factor 23 using the Edge Index, the CSRGraph and a custom implementation using only GMT.

level SHAD implementation, which, however, still provides comparable performance. This is a valuable result, especially when taking into account the substantially different development efforts. In fact, the GMT implementation has been developed in weeks, while the SHAD implementation has required about an hour. This clearly highlights that SHAD allows to quickly implement extensions and applications, with improved productivity and limited performance loss when compared to custom solutions.

V. Related Work

In previous work [5], [6], [15], we have proposed GEMS, the Graph Engine for Multithreaded System. Originally designed as a SPARQL [16] database, GEMS currently supports GraQL [17] as input query language, and adopts an attributed graph hybrid data model. GEMS and SHAD have some common design features, and in particular, the idea of abstracting the underlying details of the machine providing an abstract runtime interface. One key contribution of SHAD over previous effort, is the identification of a streamlined set of requirements imposed on the runtime system, so that SHAD can be easily mapped on a broader set of platforms. Another common characteristic of the GEMS and SHAD software stacks is that they both feature a distributed data-structure layer. However, GEMS data structures (i.e., Tables and Graphs) are specifically tailored to support GRAQL query execution over attributed graphs. SHAD's data-structures layer instead is designed to support general purpose computation and analytics workflows. As shown in this work, such data-structures can also be combined to compose higher level data-structures and libraries, such as the proposed graph SHAD

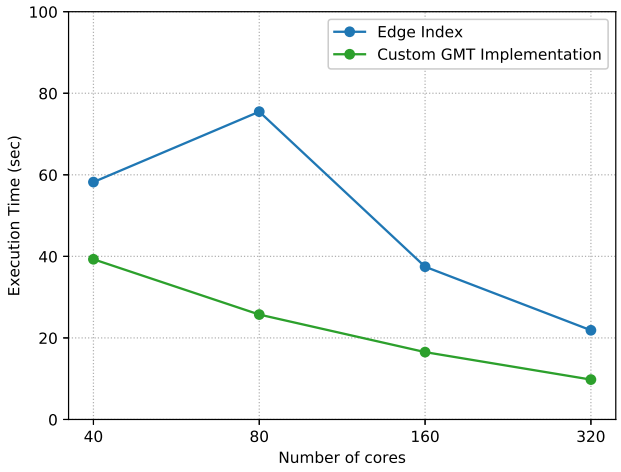

Figure 10. Detail of the strong scaling of the Triangle Counting on an Erdös-Rényi Graphs with scale factor 23 using the Edge Index and a custom implementation using only GMT.

extension.

Similar design principles are adopted in [18], which proposes a C++ template library of distributed data structures and parallel algorithms. Nevertheless, such solution adopts a fundamentally different programming model. It indeed proposes a Single Program Multiple Data (SPMD) programming model with hierarchical additions, which, from the user perspective, is closer to the MPI programming paradigm. SHAD instead offers a shared memory programming environment.

The Standard Template Adaptive Parallel Library (STAPL) [19] is a framework supporting the design of parallel programs for both shared and distributed memory parallel systems. The core STAPL library provides *pContainers* (distributed data structures) and *pAlgorithms* (parallel algorithms). *pContainers* are thread-safe distributed container that provides parallel methods that can be executed concurrently. Similarly, with SHAD data structures, *pContainers* offers a shared object view that abstract the physical data distribution. *pAlgorithms* are the parallel equivalent of STL algorithms. Similarly to STL algorithms that are written in terms of iterators, STAPL algorithms are written in terms of *views* that enable to offer multiple interfaces for the same *pContainer* (e.g., row-major or column-major view for a matrix). Intel Threading Building Blocks (TBB) [20] share the same STL philosophy of STAPL and it implements some of STAPL concepts, but it is limited to shared memory systems. SHAD instead, while providing a shared memory programming environment, overcomes this limitation.

Presto [21] proposes an extension to the R ecosystem to efficiently process large and sparse data sets on distributed systems. It introduces the concept of distributed arrays (*darray*) into the R programming language. Presto distributed arrays are partitioned using a user provided policy on creation (columns, rows or blocks) and can be reconstructed using a *split* construct. They are read-shared by multiple concurrent tasks but updates needs to be made explicitly visible to other tasks by calling an *update* function. Differently, the SHAD library advocates a more transparent and shared memory-like programming model where the not uniform data access time is hidden through fine grained multi-threading.

UPC++ [22] and Co-Array C++ [23] implement a PGAS approach. While Co-Array C++ offers a programming interface that provides a strict local-view of objects, UPC++ provides global pointers and shared arrays. SHAD extends the features offered by these approaches by providing a wider set of distributed data structures and efficient access methods while, at the same time, hiding the complexity of distributed systems.

VI. Conclusion and Future Work

In this paper we have introduced SHAD, the Scalable High-performance Algorithms and Data-Structures library. SHAD aims at facilitating the development of big data, high performance applications, by providing the users with a high-level shared-memory programming environment, and a set of general purpose data-structures with interfaces inspired by common programming languages libraries.

SHAD data-structures are thread-safe, and designed to accommodate and process significant amount of data. They can also be combined to develop additional data-structure libraries, called SHAD extensions. We have demonstrated SHAD flexibility by proposing a graph library, able to process CSR and EdgeIndex graphs, as an example of SHAD extension. On top of that, we have developed two applications, Triangle Counting and PageRank. Experimental results validate our approach, demonstrating good performance and scalability of both data-structures and user-level applications. When compared to custom, optimized applications, SHAD provides comparable performance with substantially lower development efforts.

In future work we will extend the set of SHAD extensions and supported runtime systems and architectures, so to further improve SHAD flexibility and portability. We also plan to investigate the adoption of non-volatile storage for the data-structures, so to facilitate data-recovery and fault tolerance.

Acknowledgement

This work was performed as part of the High-Performance Data Analytics (HPDA) program at the Pacific Northwest National Laboratory (PNNL). The authors would also like to thank John Feo (john.feo@pnnl.gov) who has developed and kindly shared the GMT implementation of the Triangle Counting algorithm referenced in Section IV.

References

[1] V. G. Castellana and M. Minutoli. (2018). SHAD Public Development Repository, [Online]. Available: https://github.com/pnnl/SHAD (visited on 01/23/2018).

[2] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics", *Int J. Information Management*, vol. 35, no. 2, pp. 137–144, 2015. DOI: 10.1016/j.ijinfomgt.2014.10.007.

[3] A. Putnam, A. M. Caulfield, E. S. Chung, *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services", in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, IEEE Computer Society, 2014, pp. 13–24. DOI: 10.1109/ISCA.2014.6853195.

[4] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader, "STINGER: high performance data structure for streaming graphs", in *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*, IEEE, 2012, pp. 1–5. DOI: 10.1109/HPEC.2012.6408680. [Online]. Available: https://doi.org/10.1109/HPEC.2012.6408680.

[5] V. G. Castellana, A. Morari, J. Weaver, *et al.*, "In-memory graph databases for web-scale data", *IEEE Computer*, vol. 48, no. 3, pp. 24–35, 2015. DOI: 10.1109/MC.2015.74. [Online]. Available: http://dx.doi.org/10.1109/MC.2015.74.

[6] V. G. Castellana, M. Minutoli, S. Bhatt, *et al.*, "High-performance data analytics beyond the relational and graph data models with GEMS", in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*, IEEE Computer Society, 2017, pp. 1029–1038. DOI: 10.1109/IPDPSW.2017.70.

[7] H. Kaiser, M. Brodowicz, and T. L. Sterling, "Parallex", in *ICPPW 2009, International Conference on Parallel Processing Workshops, Vienna, Austria, 22-25 September 2009*, IEEE Computer Society, 2009, pp. 394–401. DOI: 10.1109/ICPPW.2009.14.

[8] J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Active pebbles: Parallel programming for data-driven applications", in *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04, 2011*, ACM, 2011, pp. 235–244. DOI: 10.1145/1995896.1995934.

[9] D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, ser. Athena scientific optimization and computation series. Athena Scientific, 1997, ISBN: 9781886529014.

[10] J. Gonzalez, Y. Low, and C. Guestrin, "Residual Splash for Optimally Parallelizing Belief Propagation", in *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS 2009, Clearwater Beach, Florida, USA, April 16-18, 2009*, ser. JMLR Proceedings, vol. 5, JMLR.org, 2009, pp. 177–184.

[11] W. G. Macready, A. G. Siapas, and S. A. Kauffman, "Criticality and parallelism in combinatorial optimization", *Science*, vol. 271, no. 5245, p. 56, 1996.

[12] A. Morari, A. Tumeo, D. Chavarría-Miranda, O. Villa, and M. Valero, "Scaling irregular applications through data aggregation and software multithreading", in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, IEEE, 2014, pp. 1126–1135.

[13] B. Jenkins, *A hash function for hash Table lookups.* http://www.burtleburtle.net/bob/hash/doobs.html [last accessed May 2017].

[14] *The 10th DIMACS implementation challenge.* http://www.cc.gatech.edu/dimacs10/ [last accessed May 2017].

[15] A. Morari, V. G. Castellana, O. Villa, *et al.*, "Scaling semantic graph databases in size and performance", *IEEE Micro*, vol. 34, no. 4, pp. 16–26, 2014. DOI: 10.1109/MM.2014.39. [Online]. Available: https://doi.org/10.1109/MM.2014.39.

[16] E. Prud'Hommeaux, A. Seaborne, *et al.*, "SPARQL query language for RDF", *W3C recommendation*, vol. 15, 2008.

[17] D. G. Chavarría-Miranda, V. G. Castellana, A. Morari, D. Haglin, and J. Feo, "GraQL: A Query Language for High-Performance Attributed Graph Databases", in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, IEEE Computer Society, 2016, pp. 1453–1462. DOI: 10.1109/IPDPSW.2016.216.

[18] K. Fürlinger, T. Fuchs, and R. Kowalewski, "DASH: A C++ PGAS library for distributed data structures and parallel algorithms", in *18th IEEE International Conference on High Performance Computing and Communications (HPCC) 2016, Sydney, Australia, December 12-14, 2016*, IEEE, 2016, pp. 983–990. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0140.

[19] A. A. Buss, Harshvardhan, I. Papadopoulos, *et al.*, "STAPL: standard template adaptive parallel library", in *Proceedings of of SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference, Haifa, Israel, May 24-26, 2010*, ser. ACM International Conference Proceeding Series, ACM, 2010. DOI: 10.1145/1815695.1815713.

[20] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007, ISBN: 978-0-596-51480-8.

[21] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: Distributed machine learning and graph processing with sparse matrices", in *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, ACM, 2013, pp. 197–210. DOI: 10.1145/2465351.2465371.

[22] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. A. Yelick, "UPC++: A PGAS Extension for C++", in *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, IEEE Computer Society, 2014, pp. 1105–1114. DOI: 10.1109/IPDPS.2014.115.

[23] T. A. Johnson, "Coarray C++", in *7th International Conference on PGAS Programming Models*, 2013, p. 54.