Early Evaluation of Scalable Fabric Interface for PGAS Programming Models

Miao Luo[1], Kayla Seager[1], Karthik S. Murthy[1,2], Charles J. Archer[1], Sayantan Sur[1], Sean Hefty[1]

[1]Intel Corporation [2]Rice University

{miao.luo, kayla.seager, karthik.s.murthy, charles.j.archer, sayantan.sur, sean.hefty}@intel.com, ksm2@rice.edu

Abstract

Inter-processor communication is a critical factor for performance at scale. In order to achieve good performance, communication overheads should be minimized. The fabric interface library plays a major role in determining the communication overheads. This is very important for the Partitioned Global Address Space (PGAS) programming models, as these models have been designed for very low-overhead remote memory access.

The OpenFabrics Alliance has recently initiated an effort to revamp fabric communication interface to better suit parallel programming models. The new open-source interface is being called Scalable Fabric Interface (SFI). The chief distinguishing feature being that the new interfaces are being *co-designed* along with the applications that use them, such as PGAS communication libraries.

In this paper we present an early evaluation of the mapping of PGAS libraries by implementing prototypes of the popular GASNet library and OpenSHMEM over SFI. Our analysis indicates overheads of mapping to SFI are significantly lower than to the current OpenFabrics Verbs communication interface. We can reduce the number of instructions in mapping GASNet to SFI by 82%, Berkeley UPC over GASNet to SFI by 80%, and OpenSHMEM to SFI by 95% as compared to similar mappings to OpenFabrics Verbs interface.

1. Introduction

The Partitioned Global Address Space (PGAS) programming model is intended to enable emerging applications with irregular data structure and communication patterns. PGAS models provide a shared memory abstraction on distributed memory machines in order to boost programmer productivity. They also provide control of data layout and work distribution which allows application developers to take advantage of locality. These features enable very good performance and scalability. In order to deliver the desired performance and scalability, PGAS programming models require a light-weight fabric communication interface.

The OpenFabrics Alliance (OFA) provides open-source software for high-performance networking applications that demand low latency and high bandwidth. The current fabric interface offered by the OFA is Verbs. The Verbs interface originates from the InfiniBand specification, which was originally envisioned as a generic system I/O interconnect. There are many significant differences between a generic system I/O interconnect and a desired architecture that supports PGAS efficiently, as described in past keynotes [14, 23]. Regardless, various PGAS communication runtimes have been ported on to Verbs [3, 9]. However, the mapping from PGAS semantics to Verbs semantics has not been simple and straightforward due to semantic mismatches and it introduces software overhead as shown in Section 7.2.

As we head into an era of increasing integration between the processor and the fabric, it is desirable to eliminate as much software overhead caused by inefficient mappings. The OFA has created a working group, called the OpenFabrics Interfaces Working Group (OFIWG), that aims to define a fabric interface that has a tight semantic map to various applications that use it, such as PGAS programming models. Members of the PGAS communication libraries community have provided feedback in order to design the new fabric interfaces to improve the mapping of PGAS features onto fabric interface features. The new fabric interface is called Scalable Fabric Interface (SFI), and its implementation is called `libfabric`.

In this paper, we present an early evaluation of the mapping of PGAS libraries over SFI. Our motivation is to inspect the mappings to see if desired features are provided, and the software overhead (in terms of instructions executed) to go from a PGAS-level operation to a SFI-level operation. Another motivation behind this paper is to introduce SFI to the

wider PGAS community in order to elicit further feedback and encourage participation of PGAS community members in the OFIWG.

The rest of the paper is organized as follows. In Section 2, we further discuss the motivation for SFI based on the semantic match requirements of PGAS. In Section 3, we provide a brief introduction to various PGAS models that we used in this paper. Section 4 discusses related work on communication interfaces used by the PGAS community. In Section 5, we introduce SFI features in further detail and describe PGAS mappings in detail. Section 6 illustrates a prototype implementation of PGAS models over SFI. Experimental results are presented in Section 7. We summarize the conclusions of the paper in Section 8, and provide acknowledgments to OFIWG members in Section 9.

2. Motivation

In this paper we demonstrate the tight semantic match between PGAS requirements and SFI. A couple of questions arise naturally:

- Why does a good semantic match of PGAS libraries to fabric interface matter?
- Is there a benefit to providing a good semantic match if the software overhead is simply moved from a PGAS library into the fabric library?

Benefits of semantic match A good semantic match eases the tasks of developing, maintaining and providing good performance on a fabric. Any mismatches lead to the possibility of sub-optimal mapping on the fabric. For example, there are multiple GASNet implementations over InfiniBand with different performance characteristics [18]. This results in duplicated optimization efforts in multiple stacks.

As the OpenFabrics ecosystem has matured and features added (shared receive queues, eXtended Reliable Connections, etc.), the PGAS libraries have had to change significantly. Another added benefit of a good semantic match is that it provides a flexibility to update the fabric hardware with newer features and offloaded functionality while not causing disruption in the application space.

Reducing software overhead Simply creating a fabric library that implements desired application functionality is not sufficient, as it runs the risk of simply moving the software overhead from the application to the fabric library. The OFIWG is therefore carefully considering the interfaces it exposes. The included interfaces are either already implemented in hardware or are generally considered as desirable in hardware implementations.

Further, there are real-life situations where a fabric vendor can optimize desired functionality when a high-level semantic is exposed by taking full advantage of proprietary and platform specific features. It could be prohibitively expensive to customize each PGAS library for a set of hardware and platform specific environments to get the maximum possible benefits. In [25], Shainer et. al. demonstrate the benefits of this approach in the context of SHMEM and Mellanox Messaging Accelerator (MXM). The OFIWG intends to extend the benefits of such an approach to an openly developed and open-source community.

3. Background

Partitioned Global Address Space (PGAS) programming model provides a shared memory abstraction on distributed memory machines. The prominent PGAS languages include: Unified Parallel C (UPC) [26], Titanium [27], Co-Array Fortran (CAF) [21], X10 [12], Chapel [10], SHMEM [6], HPF [20], and Global Array (GA) [16]. Our work focuses on SHMEM, and GASNet.

OpenSHMEM: SHMEM (Shared Memory) [6] is a library-based PGAS programming model, which exposes a globally shared memory space for Processing Elements (PEs) to create symmetric objects and perform one-sided communication operations on them. There have been several SHMEM implementations that are specific to different commodity platforms from Cray, SGI, Quadrics, IBM [8], and Mellanox [25]. These platform-specific implementations are not portable due to minor variations in semantics and APIs.

OpenSHMEM [11] is an effort to bring a variety of SHMEM and SHMEM-like implementations into one standard specification. GSHMEM [28] is a portable OpenSHMEM implementation over GASNet communication middleware based on OpenSHMEM v1.0 specification. The reference implementation of OpenSHMEM [4] is built over GASNet. Though GASNet supports a huge range of fabric networks, it doesn't have direct atomic or collective APIs. Barrett et al. have proposed OpenSHMEM over Portals4 API [7], which has many of the scalability and performance features that were adopted by the OFIWG in the SFI definition.

GASNet: Global Address Space Networking [15] is an important communication runtime for many PGAS languages. Currently, several PGAS languages such as Berkeley UPC, Co-Array Fortran, SHMEM, Cray Chapel, and Titanium, have implementations over GASNet. It uses "conduit" system to implement portability, which allows different hardware vendors to provide hardware-specific conduits. The current version of GASNet includes support for InfiniBand, Cray XT Portals, Cray Gemini and Aries, IBM BlueGene/Q PAMI, UDP, MPI, Myrinet, and Quadrics. In this paper, we present a GASNet conduit for SFI.

4. Related Work

In this Section we discuss related work in this domain. This topic has been an active area of research. In the interest of brevity, we focus on a few relevant fabric APIs that are used for implementing high-performance PGAS.

4.1 OpenFabrics Verbs

The Verbs interface in the OpenFabrics Enterprise Distribution (OFED) is derived from the InfiniBand specification. InfiniBand was conceived of as a generic system I/O interconnect. It derived channel and Remote Direct Memory Access (RDMA) semantics from the Virtual Interface Architecture (VIA). There are several aspects of the Verbs interface that are not semantically aligned with PGAS models. These semantic gaps cause PGAS library implementations to come up with mechanisms to bridge the gaps. This results in software overhead as code must be executed by the CPU before issuing a fabric-level operation. Some of the semantic mismatches are described below.

1. **Connection-oriented model:** High-performance data paths through the OpenFabrics Verbs stack require a connection oriented model. In general, the memory footprint imposed by connection-oriented model is $O(n^2)$, where n is the number of processes in the job. There are some mechanisms to reduce the connection requirements, such as eXtended Reliable Connection (XRC), that enable an asymmetric connection mode where a sender does not need a connection to every process on a particular destination node. Neither of these modes map well to PGAS since PGAS models allow adhoc, any-to-any communication. Keeping track of connections and optimization such as on-demand connections only add to extra software overhead since such bookkeeping must be consulted every time an application issues a Put or Get operation.

   Recently Dynamic Connected Transport (DCT) has been proposed that reduces the number of connections even further. It is available in the Mellanox OpenFabrics distribution as a Verbs experimental API. DCT requires that a remote dynamic connection number be provided in the send operation. Software has to translate destination PE rank (SHMEM parlance) to a dynamic connection number. Barrett, et. al. have demonstrated that a memory lookup in the critical path can cause significant degradation for random patterns of communication [7]. This is due to the fact that the address table is usually very large since it needs to store $O(n)$ addresses. Communication with a random PE causes a lookup in a random section of the address table, leading to a cache miss. We note that communication with random PEs is a common occurrence in PGAS applications, as PGAS is designed to support irregular applications with fine-grain communication (small messages). The fabric communication is therefore stalled until the cache miss is satisfied. PGAS models really require a reliable connection-*less* and *logically* addressed transport. In Section 5, we describe the logical addressing support in SFI.

2. **Unreliable Datagram:** InfiniBand Verbs provide a connection-less model that enables unreliable datagrams. While the connection-less model is beneficial for scalability purposes, it suffers from performance issues. It only supports messages of one MTU (Maximum Transmission Unit, typically 4KB), and is unreliable. Therefore, without zero-copy assistance, software has to implement fragmentation, re-assembly and re-transmission. This results in very significant performance penalties, and therefore not appropriate for use in implementation of PGAS models. In Section 5, we describe the Reliable Datagram Messages model in SFI that enables fast, connection-less transport protocols, and enable zero-copy transfers.

3. **Memory registration and Keys:** Memory registration is required in InfiniBand for all buffers that are used in communication operations. It usually implies two different operations: (a) pinning of pages in memory, and (b) translating virtual addresses into physical addresses. This is a mismatch for global address-space models that only require communication from virtual addresses. PGAS runtimes have to adapt to bridge this mismatch. Bell and Bonachea show such a mapping in [9] for GASNet. It is to be noted that the software overheads (such as checking the cache, and managing keys) remain even though a majority of HPC applications fit their workloads in physical memory. Recently, on-demand paging has been proposed as an extension to the Verbs API. Using this feature, the entire virtual address range can be registered once (even though pages are not allocated). This definitely improves the mapping to PGAS, although it is still not ideal. Every process has a different memory region object with potentially a different remote access key, which is necessary for the remote processes to access these memory regions. Therefore, remote keys need to be exchanged between processes before any remote memory access can occur. These keys need to be looked up when initiating communication with a remote process from the relatively large array. As mentioned earlier in this section, this could result in a cache miss for certain communication patterns that stalls the fabric operation until it is satisfied.

4.2 OSU MVAPICH2-X

There is increasing interest in a hybrid programming model, such as MPI+PGAS for Exascale computing. MVAPICH2-X [3] provides such a unified high-performance runtime for hybrid programming, that is customized for InfiniBand clusters. It currently supports UPC and OpenSHMEM programming models. The unified runtime also offers other benefits like reduced resource consumption due to sharing of all network related resources. Currently, MVAPICH2-X is a binary only release.

MVAPICH2-X utilizes OpenFabrics Verbs in a high-performance and scalable manner. It faces the challenges described in the previous section 4.1 in order to map PGAS semantics to Verbs. In this paper, we measure the software overhead in MVAPICH2-X caused by this mapping. Since

MVAPICH2-X is available only in binary form, we cannot provide source-code level analysis. We measure software overhead using dynamic binary instrumentation tools that allow us to generate an instruction trace.

4.3 Mellanox ScalableSHMEM and MXM

ScalableSHMEM and MXM have been co-designed to overcome scalability issues to improve efficiency. Shainer, et. al. present the co-design of communication libraries with the underlying hardware interconnect solution approach in [25]. The authors were able to demonstrate a 50x improvement in performance and scalability by adopting this approach. We agree with the authors that there is much performance lost in the current OpenFabrics Verbs and its mismatch with the PGAS communication requirements. We listed some of the mismatches previously in this section 4.1. Therefore, such a co-design approach shows enormous benefits.

The MXM communication interface and ScalableSHMEM are Mellanox proprietary and closed-source (binary only releases). Mellanox is a current member of the OpenFabrics Interfaces Working Group (OFIWG) that is developing SFI.

4.4 Sandia Portals4

The Sandia Portals interface, has been used in this domain since the early 90s. Portals allowed the user to describe actions that need to occur on memory segments. The underlying kernel or hardware would be able to place incoming data directly. Users could build communication middleware using the Portals data-structures as building blocks. Many PGAS runtimes have been ported to use Portals, such as GASNet [7, 13].

Portals4 [5] adds light-weight non-matching interface to boost PGAS messaging rates. Additionally, it introduces logical rank-based addressing to simplify code paths and eliminate cache misses. Members of the Portals team from the Sandia National Laboratories are also participating in the OFIWG effort to craft the SFI interface.

4.5 Cray uGNI/DMAPP

The user Generic Network Interface (uGNI) is an API from Cray Inc. released along with the Gemini network. It is useful for implementing MPI on the Cray XE system. The Distributed Shared Memory Application (DMAPP) API is specifically designed for PGAS applications. The DMAPP API exposes very low level operations well suited for SHMEM. The design point of uGNI is to simply expose the communication capabilities of the Gemini router ASIC. As such, this results in a low level interface that does not aim to be a high-level semantic match to MPI, although allowing an efficient implementation of MPI on it.

The uGNI API provides logical endpoints to improve MPI scalability. The Node Translation Table (NTT) allows upper-level software to specify the group size, job id, protection tag and other unique identifiers. The API does not require upper-level software to enumerate all possible end point ids. The logical endpoints can be bound to remote endpoints via NTT index.

The GNI/DMAPP are proprietary Cray APIs that are available on Cray systems. Cray is actively participating and Paul Grun (Cray) is the co-chair of the OFIWG effort.

4.6 IBM PAMI and APGAS Runtime

The IBM Parallel Active Messaging Interface is a common messaging interface for all IBM HPC platforms. It is also extensible to other networks, such as InfiniBand. It supports a broad range of programming models, such as SHMEM, MPI, etc. and applications can also directly use PAMI. The Asynchronous Partitioned Global Address Space (APGAS) Runtime [17] is a library that is specifically used by PGAS languages such as UPC, X10 and CAF in the IBM environment. PAMI focuses on latency, throughput optimization and provides the flexibility to the application to apply threads to communication contexts, thereby maximizing the parallelism inside the communication library. Experience with PAMI is extremely relevant to SFI, as it has revealed the benefits of a communication runtime that can support multiple programming models in a high-performance manner.

PAMI and APGAS are proprietary to IBM and were developed internally. PAMI is available as open-source, and is typically deployed on IBM systems. IBM is also participating in the OFIWG effort, which has an open-design/development and open-source approach.

5. Scalable Fabric Interface (SFI)

In previous sections, we discussed the issues mapping PGAS to current OpenFabrics Verbs. In this section, we summarize the important features of the Scalable Fabric Interface (SFI). We highlight the features that mitigate these semantic mismatches. The main goal of SFI is to provide a high-level semantic match for different requirements of applications with minimal software overhead, while providing portability. The instantiation of the SFI in a software library is called *libfabric*, the source code of which may be obtained from [2].

The implementation of SFI (i.e. libfabric), consists of two distinct parts:

1. A set of fabric providers that implement the communication interfaces for a particular fabric hardware.
2. A general purpose framework that provides a plugin-like capability for providers.

The general usage flow is as follows. Fabric providers that implement functionality register with the SFI framework for discovery. They advertise the features and capabilities they offer. The application (in this case the PGAS communication runtime), specifies the features and operations it desires from the fabric library. The framework then searches for matching providers, and if found, delivers the matching provider information to the application. If there are multiple providers,

they may be presented in rank order, where the rank order is determined by the system administrator when installing SFI. The application then uses this information to allocate fabric objects such as endpoints, etc. Alternatively, the application can also instruct the framework to open a specific provider.

The framework component and communication operations are co-designed in a way that they are cohesive and not simply a union of all the fabric interfaces available today. The framework is designed to be extensible for new programming models, hardware and networking capabilities.

5.1 SFI Architecture

SFI follows an object-oriented model and the various objects used in communication operations are related to each other. In this section, we provide a brief description of the SFI architecture. Readers are encouraged to refer to the programming manuals, or man pages that are available with the SFI distribution [2].

The SFI architecture is shown in Figure 1. The interfaces are grouped by types - interface control and communication operations. The interface control operations are used to control the fabric interface which typically refers to a physical or virtual NIC. The communication operations represent the style and semantics of the communication that are desired by the application.

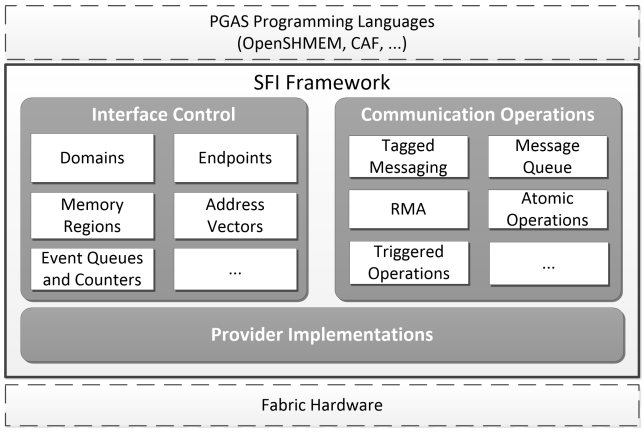

Figure 1. Architectural View of Scalable Fabric Interface Framework

Domains A domain defines the boundary for associating different resources together. Fabric resources belonging to the same domain may share resources. A domain typically refers to a physical or virtual NIC or hardware port; however, a domain may span across multiple hardware components for fail-over or data striping purposes.

Endpoints An endpoint is a transport level communication portal. Endpoints are associated with access domains and can perform data transfers. Endpoints may be connection-oriented or connection-less, and may provide data reliability. PGAS models map very well to the reliable datagram messages (RDM) type endpoint. This type of endpoint does not require $O(n)$ memory and facilitates adhoc any-to-any communication model.

Memory Region Registered memory regions associate memory buffers with permissions granted for access by fabric resources. A memory buffer must be registered with a resource domain before it can be used as the target of a remote RMA or atomic data transfer. It is also possible to register any range of addresses in the virtual address space, whether or not those addresses are backed by physical pages or have been allocated to the application. This facilitates communication using virtual addresses. Memory regions in SFI also allow the application to specify the key they want to associate with the region. This implies that processes in a job decide on a key *a priori* and assign it to the memory region. Since the key is known on all processes, there is no requirement to exchange the keys, thus saving $O(n)$ storage. Further, SFI enables offset-based addressing (by matching on keys associated with the region). Offset-based addressing greatly simplifies and enhances scalability for distributed arrays. The origin process does not need to store the base address of the start of the region at the target. Rather simply specifying a key associated with the distributed memory region and the offset is sufficient. This is in contrast with OpenFabrics Verbs that do not allow the application to choose a key for a memory region, and having to track the base address at each target process.

Address Vectors Addresses are required for communication between endpoints. For high-performance, addresses must be resolved such that route specific information, such as Service Levels, Path MTU etc. are known before the send or write calls. Therefore, every remote address represents some local object that must be managed on the initiator. At scale, management of these address tables is critical as this is an $O(n)$ structure. For example, the address table may be shared within a node, or even managed by hardware. It may be desired that the addresses are simply referred by their index, as opposed to the entire address. SFI Address vectors provide these functions.

Event Queues and Counters Event Queues and Counters interfaces allow the applications to check or wait until the completions of asynchronous operations. Event queues are used for obtaining complete information for a communication operation when it completes. SFI separates error events into a separate event queue thereby simplifying the path to handle successful completions. The error event queue only needs to be polled when an error is encountered. Counters, on the other hand are useful for obtaining aggregate information on completions. Counters provide a light weight completion event that uses a small fixed amount of memory to provide simple tracking of outstanding communication requests. This optimization is desirable for PGAS libraries.

Tagged Messaging This allows communication operations to carry a tag, that is matched on the receiving end to choose

the particular receive buffer in which data is delivered. This is intended to be used primarily by MPI.

Message Queue This allows simple messaging operations without any tag. This is similar to the channel semantics offered by OpenFabrics Verbs. The message queue semantics combined with the Reliable Datagram Messages type endpoint offer very scalable any-to-any messaging. The message queue model is useful for Active messaging, as provided by GASNet.

Remote Memory Access (RMA) RMA operations are used to transfer data directly between a local data buffer and a remote data buffer. RMA transfers occur on a byte-level granularity, and no message boundaries are maintained. The RMA operations are primarily designed to suit PGAS programming models.

Atomic Operations Atomic transfers are used to read and update data located in remote memory regions in an atomic fashion. SFI defines a wide variety of atomics, as compared to OpenFabrics Verbs that defines only 64-bit wide atomics. Datatypes and operations supported by SFI atomics are listed as follow:

- **Datatypes:** int8, uint8, int16, uint16, int32, uint32, int64, uint64, float, double, float complex, double complex, long double, long double complex
- **Operations:** minimum, maximum, sum, product, logical OR, logical AND, bit wise OR, bit wise AND, logical XOR, bit wise XOR, read, write, compare and swap, compare and swap not equals, compare and swap less than or equal, compare and swap less than, compare and swap greater or equal, compare and swap greater than, and bit wise masked swap

5.2 Mapping PGAS over SFI

A list of PGAS requirements was created and presented at the OFIWG by Howard Pritchard after consultation with many members of the PGAS community. The original presentation can be found at [24], and a complete list of the requirements brought forth by the community is available from [1]. In this section, we present a review of the PGAS requirements and how they are mapped onto SFI. This is a list of high-level requirements from several PGAS models/libraries. Table 1 summarizes some of the critical requirements and SFI mappings.

Previously, in this section we have described the scalability features for reliable connection-less endpoints, rank enumerated addresses, memory registration and atomic operations. We provide some details about the remaining aspects that are described in Table 1.

Remote completion Remote (or global) completions are very important for PGAS models for implementing strict or relaxed memory models. At certain points, the compiler/library needs strong guarantees that any writes have indeed been reflected in target memory. For the sake of efficiency, it is highly desired that this synchronization overhead be as small as possible and not require further CPU involvement at target. Additionally, various networks may implement this in a different manner and leverage internal hardware acknowledgments. The PGAS libraries desire a clean method by which this semantic requirement is expressed.

In SFI, completions can be observed either through an event queue, or a counter. SFI offers various properties at the endpoint-level to specify what a completion means. By default, a completion implies that the operation completed locally - i.e. the initiator is free to release resources associated with the operation. In addition, SFI provides a flag `FI_REMOTE_COMPLETE`, that can be used to configure an endpoint. When this flag is configured at both initiator and target endpoints it indicates that a completion should not be generated until the operation is complete at the target.

Local buffer reuse OpenSHMEM exposes blocking write operations that allow the reuse of the send buffer when the call returns. Since these operations are used for very small messages, it is important that there be a low overhead mechanism by which small messages are injected into the network and buffer can be reused. SFI provides a direct mapping to this requirement by the `fi_inject` function.

Ordering semantics PGAS models offer a wide variety of ordering semantics - from relaxed to strict consistency. PGAS libraries and compilers face a tough task in providing ordering semantics in a portable manner. Every platform and network has different ordering semantics, and often it is exposed at a very low level. SFI offers a wide variety of ordering semantics such that the PGAS compiler or library can request a particular order from the network (as opposed to implement it above the network layer). This allows the SFI provider to expose all optimizations since it is aware of the order required by the application.

Piggyback data It is often useful to send metadata along with a message. Such as the data associated with a remote function invocation and the arguments. Another use case is providing a pointer to a remote structure that is associated with the data delivered (say a pointer to a request structure). SFI exposes up to 64-bits of immediate (or piggybacked) data that can be sent along with a message.

Minimizing instructions SFI is designed from the ground up to minimize instructions executed in the critical path. It intends to fully utilize compiler optimizations that are possible by inlining methods and inter-procedural optimizations (IPO). In the current OpenFabrics Verbs, these compiler optimizations are not possible. This is due to the fact that provider functions are called through function pointers, thus making it *very* hard, if not impossible for the compiler to do advanced analysis. SFI defines a direct provider method (`FI_DIRECT`), that allows a version of libfabric to be installed that has the provider built in. In this mode, the appli-

PGAS Requirement	SFI Features and Capabilities	Description and Impact
Scalable endpoint memory usage	Reliable connection-less endpoint	$O(1)$ endpoint memory usage and no lookups
Low overhead endpoint enumeration	Address Vectors	Logical addressing of endpoints by PE/rank
Scalable memory registration	Application-specified Remote Keys	$O(1)$ storage and no lookups
Sparsely populated memory regions	Register entire virtual address range	Registrations no longer in critical path
Remote Completion for Put and Get	Remote completion attribute for endpoint	No communication required to ascertain whether writes completed at target
Local buffer reuse for small writes	Message injection with immediate local completion	No polling required to detect local completion
Flexible ordering semantics	Allows app to specify Message ordering and Data ordering (RAR, RAW, WAR, WAW, RAS (read-after-send), WAS, SAR, SAW, SAS)	Efficient code paths where required ordering matches fabric ordering. Reduction in code complexity where emulation is required (emulation can be implemented in the provider)
Rich set of Atomic operations	Supports 14 different datatypes and 19 different ops	Enables wide variety of optimization and synchronization algorithms
Sending metadata along with messages	Supports provider configurable amount of metadata to be sent along with messages	Increased metadata available to applications. An example: passing target side pointers (when provider supports 8B metadata).
Minimize instructions in critical path	Supports a direct-mode where an SFI provider can present inline versions of critical path functions	High messaging rate and lower latency

Table 1. Mapping of PGAS Requirements to Programming Languages

cation links directly with libfabric and the provider, allowing full use of inlining and IPO. The resulting binary works only on the target provider - but that is a common usage model in High Performance Computing (HPC) area, as there is often only one high-performance fabric installed on the system.

6. Implementations of PGAS Libraries over SFI

In this section, we examine the implementations SHMEM and GASNet over SFI.

6.1 SHMEM over SFI

We implement the OpenSHMEM 1.0 standard over SFI. Our prototype is derived from the SHMEM-Portals implementation. The design is optimized by using several key features of SFI. These features include reliable connection-less endpoints, a light-weight counter based completion mechanism, and very low overhead mappings for communication critical operations like small Puts.

SHMEM-SFI communication is centered around a single endpoint assigned to each PE as illustrated in Figure 2. The endpoint encapsulates the settings and resources for the desired SHMEM communication scheme. Endpoints are logically addressed, i.e. address by PE number, via an SFI address vector table that is bound to the endpoint. We expose the full address space through our endpoint and directly use remote virtual addresses in Put/Get calls. We use three counters for synchronization that are bound to the endpoint. In addition, a SFI memory region is bound to the endpoint to count inbound Puts and Gets.

We use the atomic and the RMA APIs on the endpoint. Synchronization is achieved by using three counters: (a) outgoing writes counter, (b) outgoing reads counter, and (c) incoming/remote reads and writes counter. The outgoing write counter is used to implement `shmem_quiet` semantics. The outgoing read counter is used to force an internal wait on get, swap, and fetch. The incoming reads/writes counter is used to implement the `shmem_wait_until`.

Small write messages (such as `shmem_int_p`) take advantage of SFI's message "inject" feature which ensures local buffer availability once the call returns as mentioned in Section 5.2. This maps to SHMEM put semantics. Reads, and atomics wait for remote completion regardless of mes-

sage size. The medium-sized write messages uses a bounce buffer to enable buffer reuse. The larger messages are fragmented and sent directly (not buffered). Both medium and large messages use a queue to track completion. Once the queue is full or the non-buffered fragments are in flight, an internal wait routine for queue completion has to block until the fragment Put is completed. This is to ensure safe send buffer reuse. All writes, regardless of size, update the counter since it makes implementation of fence and quiet very simple.

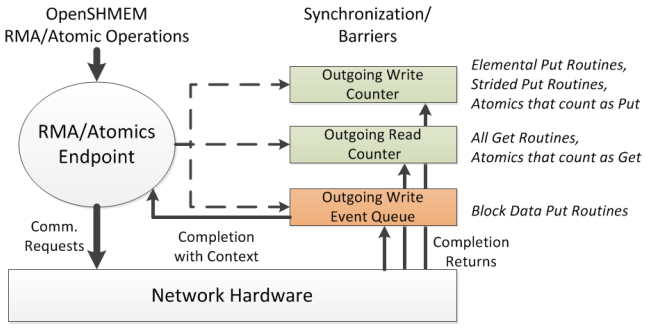

Figure 2. OpenSHMEM design over SFI: a single endpoint is used for RMA/Atomics operations, with Event Queue for block put routines and Counters for other operations.

6.2 GASNet SFI Conduit

GASNet specification defines two groups of APIs: 1) The Core APIs, which include job control interface, active messaging interface, and atomicity control interface; 2) The Extended APIs, which include memory-to-memory data transfer functions, register-memory operations, and barriers. In this section, we explain how to utilize SFI-based conduit to support GASNet Active Messaging and data transfer APIs with minimal software overhead. The basic structure of GASNet SFI conduit is shown in Figure 3.

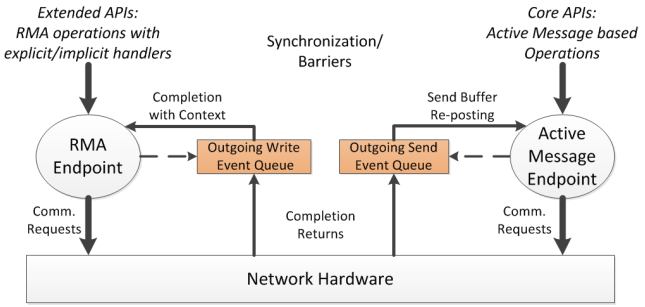

Figure 3. GASNet over SFI: two endpoints for RMA operations and Active Message based operations, respectively, with separate Event Queues for completions.

Active Messaging APIs in GASNet handle three categories of active messages: short, medium, and long. Short active messages carry only arguments. Medium active messages include a data payload in addition to the arguments. Long active messages include a data payload with predetermined address on the remote node, in addition to the arguments. In order to handle the Active Messaging communication requirement from GASNet, we open an SFI Endpoint with Message Queue capability (AM Endpoint). As introduced in Section 5.1, Message Queue is able to deliver simple messaging operations without any tag. In order to receive active messages from any remote PE, the users need to pre-post buffers on this AM Endpoint. We use the `FI_MULTI_RECV` feature in SFI to implement efficient receive operations. The multi-receive feature allows the application to specify a large area of buffer space *once*, and the underlying fabric implementation places incoming data into this buffer as it arrives. Thereby, saving the application from the complexity of actively managing the receive buffer. Once the available buffer space is used up, an special event which contains information about the consumed buffer is generated and returned to the users through the Event Queue. New incoming messages are received into the next free buffer. If all the pre-post buffers are consumed before the user can post any more, incoming data can be dropped. The OFIWG is currently discussing support for exposing such flow-control events. This feature is similar in spirit to the persistent match entry in Portals4. We set the `FI_MULTI_RECV` flag when opening the AM Endpoint. The sending process of active messages in GASNet SFI conduit varies between different categories of the Active Messaging APIs. For short and medium active messages, we use a temporary buffer to pack all the arguments and data payload. Then the send buffer are posted to the AM Endpoint through either `fi_injectto` or `fi_sendto` APIs, depending on the total size. If the message is sent through `fi_injectto`, SFI guarantees the send buffer can be reused after returning from this API. This is similar as the "inject" feature introduced in Section 6.1. However if the message is sent out through `fi_sendto` API, these temporary buffers need to be freed or returned to send buffer pool depending on different implementations. Thus we attach an outgoing Event Queue to the AM Endpoint. After a local completion is achieved for a send request with `FI_EVENT` flag, SFI generates a notification event and enqueue it into the active message Event Queue. This event contains required information to identify and process the send buffer. For long active messages, the parameters are sent through AM Endpoint, while the data payload are sent through RMA Endpoint, as introduced below.

For the memory-to-memory data transfer functions from Extended API, we create the second SFI Endpoint, which is represented as RMA Endpoint in the figure. Virtual addresses are used for write/read on the RMA Endpoint in GASNet similar to SHMEM/SFI. The SFI APIs used are: `fi_inject_writeto`, `fi_writeto`, and `fi_readfrom`. GASNet RMA data transfer APIs include blocking memory-to-memory transfers and non-blocking memory-to-memory transfers. GASNet defines two types of asynchronous com-

munication operations depending on whether they return a handle or not. The explicit handle operations return a handle that is used to synchronize the specific operation in flight. The implicit handle operations do not return a handle, and the synchronization is accomplished by calling a synchronization routine that synchronizes all outstanding operations. For blocking RMA operations or non-blocking RMA operations with explicit handles, GASNet requires a completion on a certain outgoing communication request. Event Queue is necessary for such completion requirements. Although the synchronization functions for implicit-handle non-blocking RMA operations can wait for a set of outgoing requests, these functions should only synchronize implicit-handle operations. In order to fulfill the requirements of the GASNet specification, we utilize Event Queue for RMA Endpoint as well. This RMA Event Queue should be polled only if there are outstanding RMA operations exist. By recording the number of outstanding RMA operations, we can reduce the overhead in network progress engine by waiting until those are complete. We are currently exploring the possibility of utilizing SFI counters in the GASNet design.

7. Results

In this section, we present the results of our evaluation mapping PGAS libraries to SFI, including OpenSHMEM, GASNet, and Berkeley UPC over GASNet. As the SFI stack is still under definition, we think an analysis that focuses on the impedance match from PGAS to SFI is timely. We measure the semantic alignment by counting the number of instructions required to translate a PGAS operation into an SFI operation. It is better to have fewer instructions as it indicates a tight semantic fit. This provides a good incentive to any SFI provider to highly optimize the implementation (whether through hardware or software means) as a way to optimize the performance of that particular PGAS library. This also insulates the PGAS layer from excessive code modification and changes when being ported across multiple vendors and multiple generations of hardware.

7.1 Experimental Environment

The evaluation is done on two dual socket nodes containing Intel® Xeon® X5570 Quad-core Nehalem CPUs, running at 2.93GHz with 12GB of host memory. The nodes are connected by a Mellanox QDR/10GigE ConnectX InfiniBand HCA (MT26428). The nodes are running Red Hat Linux 6.5, with kernel version 2.6.32-431.el6.x86_64. The Intel® C++ Composer XE 2013 SP1 (Intel® C++ compiler 14.0) is used with "-O3 -ipo" flags. GASNet 1.22.4 and MVAPICH2-X 2.0 are utilized for the evaluation. We use a variation of the Intel® Pin for generating instruction traces for accurate measurement.

In order to focus on the mapping overhead, we choose to measure the instructions spent only in the fabric communication middleware. Thus as shown in Figure 4, we measure the instruction counts between the PGAS applications and the providers for the fabric library APIs (inside the dashed line rectangle). As a concrete example for the SHMEM prototype, we measure the instructions from `shmem_int_p` to SFI function `fi_injectto`. We compare these instructions with other implementations of SHMEM over OpenFabrics Verbs by measuring instructions from `shmem_int_p` to Verbs API `ibv_post_send`.

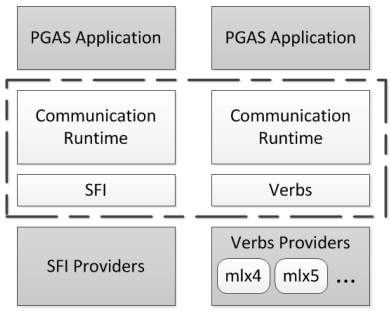

Figure 4. Scope for Instruction Count Measurement

7.2 Impact of Software Overhead on Performance

In this section we present data to support our claim that software overhead has an impact on small message latency and messaging rate. We evaluate the effect of software overhead in existing PGAS runtimes based on current Verbs communication interfaces. We use the MVAPICH2-X runtime for this experiment. We measure the Cycles per instruction (CPI) using the Oprofile system profiling tool available with RedHat distribution. We capture two events, `CPU_CLK_UNHALTED` and `INST_RETIRED`. The ratio gives us CPI. In order to measure these metrics correctly, we completely occupy one CPU core, such that the messaging library code alone executes on the core. Further, to ensure that the critical path (send operation) instructions are measured, we repeatedly execute *only* the send path, by pumping many messages of size 1B, window size=64 and 1M iterations.

The latency of executing the particular code path on the CPU core is given by the following formula:

$$time = \frac{INST\_RETIRED \times CPI}{CPUCorefrequency}$$

We measure the CPI to be 1.1 on our platform. Figure 5 shows the software overhead in terms of time. We present analytical model data with different CPU core frequencies and CPI. We observe that the software overhead is in excess of 200ns for core frequency of 1GHz - 1.8GHz. Power required for a CPU is directly related to a frequency. Scaling frequency to improve software overheads is not a good strategy, since power limitation is the biggest challenge going to Exascale.This indicates that software overhead is very significant factor for next generation Exascale CPU cores.

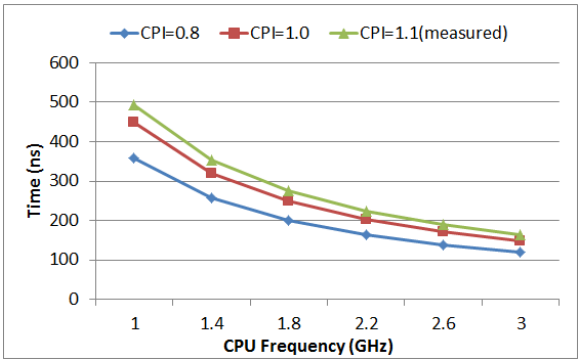

Figure 5. Estimated Latency Overhead

7.3 Instruction Count Comparison

7.3.1 OpenSHMEM

We first compare the software overhead of prototype OpenSHMEM over SFI implementation with two popular OpenSHMEM implementations: OpenSHMEM reference implementation over GASNet IB Verbs conduit (GASNet-IBV) and MVAPICH2-X software package (MV2X). We also include OpenSHMEM reference implementation over GASNet SFI conduit (GASNet-SFI). We use SFI in the direct mode, as described in Section 5.2 to enable full compiler optimizations, such as inlining and IPO. We measure the instruction counts for two representative OpenSHMEM semantics: `shmem_int_p` and `shmem_quiet`.

The results are shown in Figure 6. SHMEM-SFI stands out as the thinnest implementation for the Put communication call at 16 instructions. MV2X shows the largest instruction count (402), followed closely by GASNET-IBV (388). Both implementations show a significant gap between the two SFI implementations: GASNet-SFI and SHMEM-SFI. SHMEM-SFI is able to use fewer instructions compared to other implementations in `shmem_int_p` due to its memory mapping scheme. SHMEM-SFI exposes the full address space at initialization, and benefits from an inherently symmetric address space across PE's with minimal cost to the Put call path. All SHMEM-GASNet implementations have an added overhead of the segmented address space look-up for the symmetric heap utilized in the SHMEM-GASNet mapping. The burden of the segmented addresses look-up is most noticeable in GASNet-SFI given the smaller number of overall instructions; it accounts for about 25% of the overhead. GASNet-SFI shows an additional instruction overhead when compared to SHMEM-SFI due to its implicit network progress requirement.

GASNet-IBV and MV2X implementations have further overheads in their memory mapping scheme that account for the majority of the instruction count gap between the SFI implementations. For instance, the GASNet-IBV implementation dynamically pins address space segments for each Put call. This accounts for almost 90% of the overhead associated with the instruction count. MV2X on the other hand uses registration-cache to look up registrations keys for addressing which adds a significant overhead in the call path.

We observe a similar trend in Figure 6 for the quiet routine instruction counts. The two SFI implementations continue to show smaller instruction counts compared to GASNet and MV2X implementations. The SHMEM-SFI implementations instruction path benefits significantly from avoiding the complicated progress engines which are required for GASNet and Verbs interfaces. SHMEM-SFI's quiet routine is only 11 instructions because it solely relies on a wait for the Put counter. GASNet-SFI on the other hand has additional overhead from polling an RMA queue. The instruction gap between GASNet-SFI and MV2X and GASNet-IBV is from the latter two implementations employing multiple rounds of polling for the receive and send progress engine.

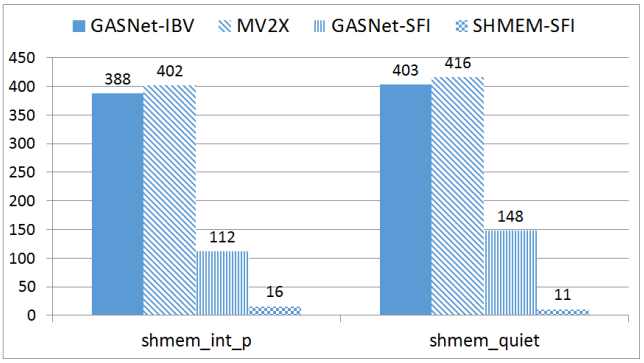

Figure 6. Instruction Counts for Different OpenSHMEM Implementations

7.3.2 GASNet

In Section 7.3.1, we already present the performance of GASNet SFI conduit in the environment of OpenSHMEM. In this section, we further present three GASNet operations which are not being utilized in previous examples, including `gasnet_AMRequestShort`, `gasnet_put_nbi`, and `gasnet_wait_syncnbi_put`. `gasnet_AMRequestShort` is one of the core APIs for short Active Message operation. In prototype GASNet SFI conduit, as described in Section 6.2, Active Message communication requests are posted to an AM Endpoint. `gasnet_AMRequestShort` function maps to `fi_injectto`. Temporary buffer managements for short message injection are left for each provider to fully optimize according to the hardware features. Comparing to IB Verbs conduit, SFI conduit reduces the instruction counts from 345 to 106, as illustrated by Figure 7.

On the other hand, `gasnet_put_nbi` belongs to extended APIs for memory-to-memory direct access operations. `gasnet_put_nbi` function launches non-blocking implicit put request while `gasnet_wait_syncnbi_put` blocking waits all outstanding non-blocking implicit Put requests. Similar to the evaluation for `shmem_int_p`, the in-

struction counts for RMA Endpoint requests are reduced by about 300. For synchronization API `gasnet_wait_syncnbi_put`, we add a counter for outstanding RMA operations, in order to save unnecessary polling overhead for RMA Endpoint. As a result, we are able to reduce the instruction counts for the wait function from 160 to 56.

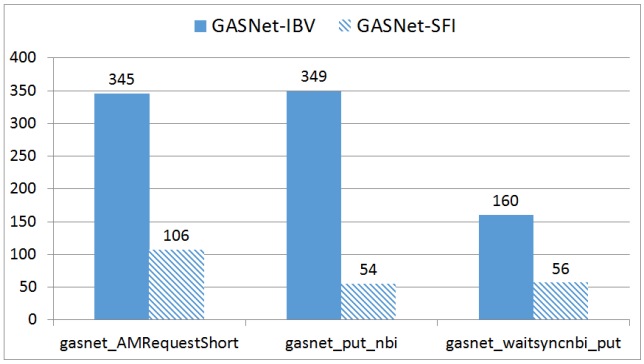

Figure 7. Instruction Counts for GASNet Conduit

7.3.3 Berkeley UPC

Berkeley UPC is an open-source implementation of UPC specification based on GASNet communication runtime [19]. We evaluate the impact of the GASNet SFI conduit in UPC environment in this section. Figure 8 shows the instruction counts for a write operation on an int type shared variable on a remote node and the synchronization function `upc_fence`. The remote shared variable access operation measured is listed in Pseudocode 1:

Pseudocode 1. Shared Variable Access

```
shared int x[THREADS]; // global shared array
test () {
    int y; // local variable
    y = ...;
    if(MYTHREAD==0) {
        x[1] = y; // instruction counts measured
    }
}
```

For the remote shared variable access operation, the size of the data is usually equal to an element size such as sizeof(int), The Berkeley UPC translator translates this access into a Put operation, which finally maps to a blocking memory access function in GASNet: `gasnet_put_bulk`. GASNet IB Verbs conduit invokes a RDMA registration strategy, which is called `firehose` [9], in order to get prepared for the small RDMA Put operation. The *firehose* algorithm is necessary for pinning-based network APIs such as Verbs. It helps reducing the host-level synchronization overhead for small Put/Get operations. According to the profiling results, 44% instruction counts are spent in *firehose* for remote memory pin-down and local pin-down check. Other than that, 47% instructions are spent in handling different completion requirement and choosing different Put algorithm. On the other hand, SFI conduit directly map the GASNet API `gasnet_memput_bulk` with SFI RMA operation `fi_inject_writeto` through the RMA Endpoint, as introduced in Section 6.2. By utilizing Event Queue, SFI conduit is able to get rid of the overhead in managing completion mechanism in the communication runtime level. The completion mechanism is left for the providers or even hardware to optimize. GASNet SFI conduit can reduce the instruction counts of IB Verbs conduit by 80% for remote shared variable access and 50% for `upc_fence`.

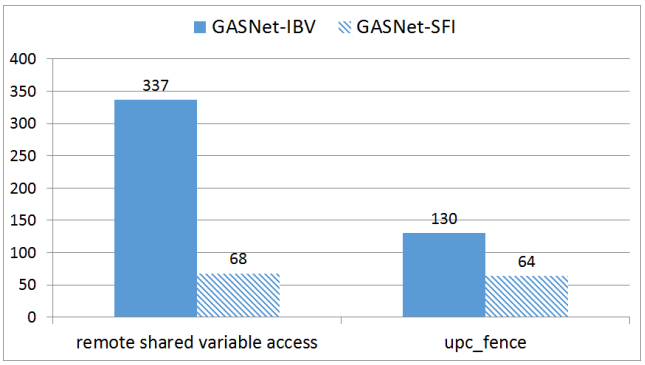

Figure 8. Instruction Counts for Berkeley UPC over GASNet Conduit Comparison

8. Conclusions and Future Work

In this paper, we provided an initial evaluation of the mapping of PGAS to the Scalable Fabric Interface. Our evaluation reveals that SFI is a better semantic fit for PGAS programming models. We illustrated two prototype implementations over SFI: OpenSHMEM and GASNet communication runtime. We evaluated the software overhead of the three prototype implementations over SFI by calculating instruction counts between PGAS language semantic APIs and hardware library, including OpenSHMEM, GASNet, and Berkeley UPC. Comparing with GASNet IB Verbs conduit and MVAPICH2-X, which are designed for the current OpenFabrics Verbs communication interface, prototypes over SFI can significantly reduce the instruction counts in critical path. GASNet SFI conduit can achieve 69% to 82% fewer instruction counts for RMA communication requests and 50% to 65% fewer instruction counts for synchronization operations. Furthermore, the SHMEM-SFI prototype can achieve more than 95% reduction in the critical path instructions. From the evaluation results, we can observe that SFI has the potential to solve the software overhead bottleneck in the future Exascale systems while satisfying requirements from different PGAS programming models. We would also like to notify the readers that, all the software overhead measurements in this paper are preliminary. The overhead contribution of the transport layer are omitted, which can be highly optimized by different providers through either hardware or software means.

In the future, we will continue refining the PGAS prototypes and engage the open-source community. We are also

evaluating the mapping of Co-Array Fortran to SFI. We expect that as SFI continues to develop, providers that map SFI to fabric hardware will develop and mature. When mature providers are available, we will be able to perform a full-stack analysis.

9. Acknowledgements

We would like to acknowledge the following people for their contribution to Scalable Fabric Interface in the OpenFabrics Interfaces Working Group (OFIWG). An up-to-date attendee list is available from at [22].

Paul Grun (Cray - Co Chair), Scott Atchley, Mark Atkins, Jeff Becker, Brad Benton, Frank Berry, Michael Blocksome, David Brean, Nafea Bshara, Yohann Burette, Ken Cain, Omar Cardona, John Carrier, Wendy Cheng, Barbara Collignon, Susan Coulter, Zarka Cvetanovic, Rupert Dance, Dmitry Durnov, Parks Fields, Dave Goodell, Manjunath Goentla, Rich Graham, Ryan Grant, Leonid Grossman, Ihab Hamadi, Jeff Hammond, Jimmy Hill, Kyle Hubert, Alan Jea, Christoph Lameter, Doug Ledford, Matt Leininger, Liran Liss, Patrick MacArthur, Linden Mercer, Bernard Metzler, Tzahi Oved, Geoffrey Paulsen, Steve Poole, Howard Pritchard, Tom Reu, Todd Rimmer, Hal Rosenstock, Bob Russell, Jim Ryan, Eyal Salomon, Pradeep Satyanarayana, Leah Shalev, Pavel Shamis, Bill Snapko, Dave Solt, Jeff Squyres, Tom Stachura, Nisha Talagala, Ram Vepa, Nenad Vukicevic, Bill Weber, Don Wood, Bob Woodruff.

References

[1] OFIWG requirements. `https://www.openfabrics.org/downloads/OFIWG/OFIWG-requirements.pdf`, .

[2] libfabric source. `https://github.com/ofiwg/libfabric`, .

[3] MVAPICH2-X: Unified MPI+PGAS Communication Runtime over OpenFabrics/Gen2 for Exascale Systems. `http://mvapich.cse.ohio-state.edu/`.

[4] OpenSHMEM Reference Source Releases. `http://www.openshmem.org/`.

[5] Portals 4 Specification. `http://www.cs.sandia.gov/Portals/portals4-spec.html`.

[6] R. Bariuso and A. Knies. Shmem User's guide, 1994.

[7] B. W. Barrett, R. Brigthwell, K. S. Hemmert, K. Pedretti, K. Wheeler, and K. D. Underwood. Enhanced Support for OpenSHMEM Communication in Portals. HOTI '11, Washington, DC, USA, 2011.

[8] A. Benner and T.-Y. Jea. IBM OpenSHMEM Implementation over the Parallel Active Messaging Interface (PAMI). In *OpenSHMEM 2014*, March 2014.

[9] C. Bell and D. Bonachea. A New DMA Registration Strategy for Pinning-Based High Performance Networks. In *Workshop on Communication Architecture for Clusters (CAC)*, 2003.

[10] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, (3), 2007.

[11] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem: Shmem for the pgas community. PGAS '10, 2010.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005*.

[13] D. Bonachea and P. Hargrove and M. Welcome and K. Yelick. Porting GASNet to Portals: Partitioned Global Address Space (PGAS) Language Support for the Cray XT. In *Cray User Group Meeting (CUG)*, 2009.

[14] D. Roweth, Cray. Efficient hardware support for PGAS languages. The 7th Conference on Partitioned Global Address Space Programming Models, Keynote, 2013.

[15] Editor: Dan Bonachea. GASNet specification v1.1. Technical Report UCB/CSD-02-1207, U. C. Berkeley, 2008.

[16] Environmental Molecular Sciences Laboratory and Pacific Northwest National Laboratory. The GA Toolkit. `http://www.emsl.pnl.gov/docs/global/`.

[17] M. Farreras and G. Almasi. Asynchronous pgas runtime for myrinet networks. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 4:1–4:10, New York, NY, USA, 2010. ACM.

[18] J. Jose, M. Luo, S. Sur, and D. K. Panda. Unifying UPC and MPI Runtimes: Experience with MVAPICH. In *Int'l Workshop on Partitioned Global Address Space (PGAS '10)*, October 2010.

[19] Lawrence Berkeley National Laboratory and University of California at Berkeley. Berkeley UPC - Unified Parallel C. `http://upc.lbl.gov/`.

[20] D. Loveman. High performance Fortran. *Parallel Distributed Technology: Systems Applications, IEEE*, feb. 1993.

[21] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, Aug. 1998.

[22] OpenFabrics Interfaces Working Group Attendee List. `https://www.openfabrics.org/downloads/OFIWG/OFWG_attendence_master.xlsx`.

[23] P. Kogge, University of Notre Dame. PGAS and Architecture: The Need for a Revolution. The 6th Conference on Partitioned Global Address Space Programming Models, Keynote, 2012.

[24] H. Pritchard. SHMEM/PGAS Developer Community Feedback. `://www.openfabrics.org/images/Workshops_2014/DevWorkshop/presos/Monday/pdf/11.45_2014%20_OFA_Workshop_pgas-communitry-feedback.pdf`.

[25] G. Shainer, T. Wilde, P. Lui, T. Liu, M. Kagan, M. Dubman, Y. Shahar, R. Graham, P. Shamis, and S. Poole. The Co-design Architecture for Exascale Systems, a Novel Approach for Scalable Designs. *Comput. Sci.*, May 2013.

[26] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[27] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998.

[28] C. Yoon, V. Aggarwal, V. Hajare, A. D. George, and M. B. III. GSHMEM: A Portable Library for Lightweight, Shared-Memory, Parallel Programming. In *Fifth Conference on Partitioned Global Address Space Programming Model (PGAS)*, Oct 2011.