UCX: An Open Source Framework for HPC Network APIs and Beyond

Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez
Oak Ridge National Laboratory
Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar
Mellanox Technologies
Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb
NVIDIA Corporation
Sameer Kumar, Craig Stunkel
IBM
George Bosilca, Aurelien Bouteiller
University of Tennessee, Knoxville

***Abstract*—**

This paper presents *Unified Communication X* (UCX), a set of network *APIs* and their implementations for high throughput computing. UCX comes from the combined efforts of national laboratories, industry, and academia to design and implement a high-performing and highly-scalable network stack for next generation applications and systems. UCX design provides the ability to tailor its APIs and network functionality to suit a wide variety of application domains and hardware. We envision that these *APIs* will satisfy the networking needs of many programming models such as the *Message Passing Interface* (MPI), OpenSHMEM, *Partitioned Global Address Space* (PGAS) languages, task-based paradigms, and I/O bound applications. To evaluate the design we implement the APIs and protocols, and measure the performance of overhead-critical network primitives fundamental for implementing many parallel programming models and system libraries. Our results show that the latency, bandwidth, and message rate achieved by the portable UCX prototype are very close to that of the underlying driver. With UCX, we achieved a message exchange latency of 0.89 us, a bandwidth of 6138.5 MB/s, and a message rate of 14 million messages per second. As far as we know, this is the highest bandwidth and message rate achieved by any network stack (publicly known) on this hardware.

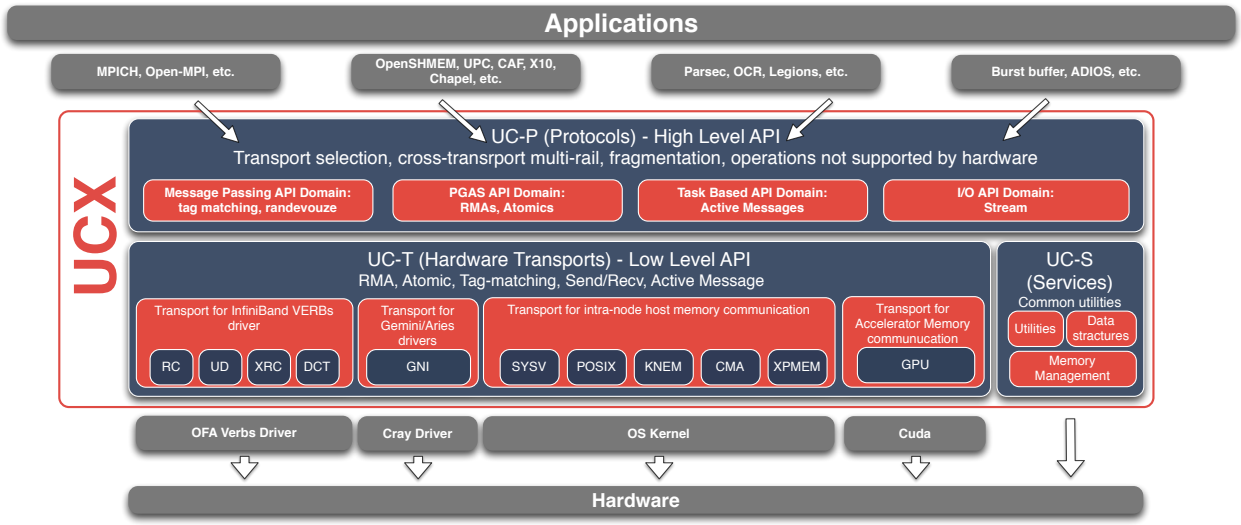
I. Introduction

Given the importance of parallel programming models for developing successful scientific applications for high-performance computing (HPC), system vendors typically provide customized network hardware along with low-level interfaces optimized to meet the functionality and performance requirements of these programming models. Some examples include Cray's Gemini network hardware with its proprietary uGNI and DMAPP interfaces; the IBM Torus network exposed through their Parallel Active Message Interface (PAMI); and InfiniBand hardware from multiple vendors which uses the Open Fabrics Alliance (OFA) Verbs and the emerging Libfabrics low-level interfaces. Because of the variety of network technology drivers in the ecosystem, it is a challenging task to design and implement portable high performance programming models that can be mapped efficiently to the current and next generation hardware. Furthermore, extending the life-expectancy of these programming models and their corresponding communication libraries over multiple generations of hardware, only deepen this challenge. This leads to a situation where a significant amount of efforts are duplicated, at all levels of the software stack, to design, tailor, and optimize higher-level communication libraries to multiple low-level interfaces. This situation can be clearly seen in many open software efforts that support multiple hardware architectures, such as with Open MPI, implemented using proprietary interfaces, including Cray's uGNI, Mellanox's MXM, and Qlogic's PSM.

There have been multiple efforts to provide a common, high-performance networking interface, but typically they are either optimized for a single programming model or limited to a particular network hardware. These efforts include GASNet [1], ARMCI [2], MXM, PSM, CCI [3], DMAPP [4], and Portals [5]. PSM and MXM are proprietary communication interfaces provided by InfiniBand vendors Qlogic and Mellanox, respectively, while the Portals network architecture is geared towards MPI but was recently extended to support PGAS based models. GASNet and ARMCI are tailored for PGAS programming models, including UPC and OpenSHMEM [6], and CCI was developed to provide a uniform interface for scientific applications' I/O needs. DMAPP was specifically

This work was supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center located at the *Oak Ridge National Laboratory* (ORNL). This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doepublicaccessplan). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Fig. 1: UCX Architecture



designed with distributed global memory as a target programming model, and supports both compiler-based (CAF and UPC) or library-based (e.g., Cray SHMEM) one-sided programming models.

To address the functionality and performance portability challenges, in this paper we present UCX, a framework for network APIs and stacks. Our approach is to unify the APIs, protocols, and implementation into a single framework, while providing the ability to tailor the functionality for a particular programming model without sacrificing efficiency. For this we are leveraging the expertise of the *University of Tennessee Knoxville* (UTK) on Open MPI and PaRSEC; the ORNL and the *University of Houston* (UH) on OpenSHMEM and PGAS languages such as Co-Array Fortran; NVIDIA's on highly multi-threading models on GPUs (CUDA, OpenACC); and IBM's expertise on MPI, OpenSHMEM, and X10 programming models and runtimes.

Besides addressing the portability challenges, UCX focuses on providing APIs and network stacks for next generation applications and emerging architectures such as systems with massively threaded nodes, heterogeneous hierarchal memories, computing accelerators (GPUs), and applications using hybrid programming models. For example, we consider the typical use case of an application employing both MPI and *OpenACC* or *OpenMP* programming models, executing on *Titan* [7] — a Cray XK7 system with AMD CPUs, NVIDIA GPUs, and a Gemini network— moving to *Summit* [8] —an IBM system with POWER CPUs, NVIDIA GPUs, and an InfiniBand network. With the unified low level UCX interface, multiple programming models — managed in their own separate runtime — can communicate between multiple devices and with a better orchestration of communication. The benefit is then twofold: the high level programming runtimes — relying on an unified communication infrastructure, — better interoperate with one another and the application, — if it itself uses UCX, — and the portability of the entire software stack is greatly improved: when new hardware is delivered, implementing the small UCX interface is sufficient to implement support for multiple programming paradigms and applications.

Another goal of UCX is to develop production grade software, while also providing a platform for researchers to innovate and continually add novel elements to the implementation. To achieve these competing goals, UCX is co-designed by researchers and engineers from national laboratories, industry, and academia. This collaboration includes ORNL, UTK, UH, Mellanox, NVIDIA, and IBM. The UCX design and implementation incorporates ideas from, and can be seen as a continuation and unification of Mellanox's MXM, IBM's PAMI, and the UCCS [9]. In the remainder of this paper, we expound on the design of UCX and present some preliminary results.

II. Design

UCX is a network API framework for modern interconnects. The goal of the API is to establish a set of interfaces for implementing multiple programming model libraries and languages that are portable, scalable, and efficient.

The framework is designed with careful consideration for emerging exascale technologies such as massively parallel computing nodes, accelerators, and hierarchical memories. The UCX architecture exposes software constructs for many-thread driven high-performance communications, communications within heterogeneous memory hierarchies, and hybrid programming models including I/O and data centric libraries. Rather than building a single interface as a "one-size-fits-all" solution, we design a framework that provides the necessary components for building various communication protocols using different levels of abstraction. Such a design delivers a high degree of flexibility enabling the implementation of new network protocols for emerging and future programming models.

For example, accelerators are expected to be a key component in exascale system design. New capabilities like GPUDirect RDMA [10] and GPUDirect Async [11] allow GPU-

based workflows to operate with minimal CPU intervention, leaving the CPU for other, more generic, tasks. The hierarchies that evolve in connecting GPUs to the node using PCIe and NVLink, and the performance characteristics they entail, need special consideration when designing a runtime for communications to/from/within GPU memory. More specifically, performance in highly-threaded environments is critical in order to realize a vision of supporting communication from inside GPU kernels [12]. UCX provides a separate transport for accelerator memory which allows implementations to be customized to the inter-GPU communication design need.

The UCX framework consists of the three main components: UC-Services (UCS), UC-Transports (UCT), and UC-Protocols (UCP). Each one of these components exposes a public API and can be used as a stand-alone library (Figure 1).

UCS is a service layer that provides the necessary functionality for implementing portable and efficient utilities. This layer exposes the following services:

- an abstraction for accessing platform specific functionality (atomic operations, thread safety, etc.),
- tools for efficient memory management (memory pools, memory allocators, and memory allocators hooks),
- commonly used data structures (hashes, trees, lists).

UCT is a transport layer that abstracts the differences across various hardware architectures and provides a low-level API that enables the implementation of communication protocols. The primary goal of the layer is to provide direct and efficient access to hardware network resources with minimal software overhead. For this purpose, UCT relies on low-level drivers provided by vendors such as InfiniBand Verbs, Cray's uGNI, libfabrics, etc. In addition, the layer provides constructs for communication context management (thread-based and application level), and allocation and management of device-specific memories including those found in accelerators. In terms of communication APIs, UCT defines interfaces for immediate (*short*), buffered copy-and-send (*bcopy*), and zero-copy (*zcopy*) communication operations. The *short* operations are optimized for small messages that can be posted and completed in place. The *bcopy* operations are optimized for medium size messages that are typically sent through a so-called bouncing-buffer. Finally, the *zcopy* operations expose zero-copy memory-to-memory communication semantics.

UCP implements higher-level protocols that are typically used by message passing (MPI) and PGAS programming models by using lower-level capabilities exposed through the UCT layer. UCP is responsible for the following functionality: initialization of the library, selection of transports for communication, message fragmentation, and multi-rail communication. Currently, the API has the following classes of interfaces: Initialization, Remote Memory Access (RMA) communication, Atomic Memory Operations (AMO), Active Message, Tag-Matching, and Collectives.

Initialization: This subset of interfaces defines the communication context setup, queries the network capabilities, and initializes the local communication endpoints. The context represented by the UCX context is an abstraction of the network transport resources. The communication endpoint setup interfaces initialize the UCP endpoint, which is an abstraction of all the necessary resources associated with a particular connection. The communication endpoints are used as input to all communication operations to describe the source and destination of the communication.

RMA: This subset of interfaces defines one-sided communication operations such as PUT and GET, required for implementing low overhead, direct memory access communications constructs needed by both distributed and shared memory programming models. UCP includes a separate set of interfaces for communicating non-contiguous data. This functionality was included to support various programming models' communication requirements and leverage the scatter/gather capabilities of modern network hardware.

AMO: This subset of interfaces provides support for atomically performing operations on the remote memory, an important class of operations for PGAS programming models, particularly OpenSHMEM.

Tag Matching: This interface supports tag-matching for send-receive semantics which is a key communication semantic defined by the MPI specification.

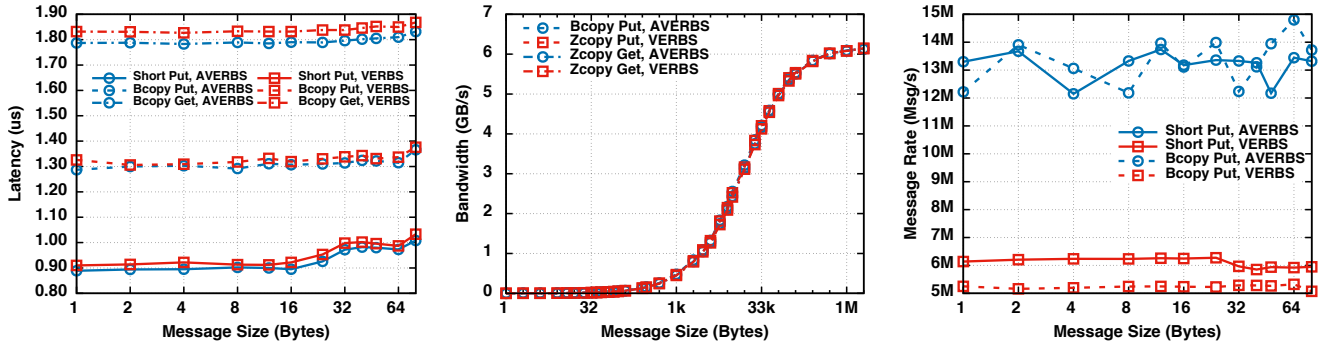
Active Message: A subset of functionality where the incoming packet invokes a sender-specified callback in order to be processed by the receiving process. As an example, the two-sided MPI interface can easily be implemented on top of such a concept [13]. However, these interfaces are more general and suited for other programming paradigms where the receiver process does not prepost receives, but expects to react to incoming packets directly. Like RMA and tag-matching interfaces, the active message interface provides separate APIs for different message types and non-contiguous data.

Collectives: This subset of interfaces defines group communication and synchronization operations. The collective operations include Barrier, All-to-one, All-to-all, and reduction operations. When possible, we will take advantage of hardware acceleration for collectives (e.g., InfiniBand Switch collective acceleration).

III. Results

This evaluation demonstrates preliminary functionality of the UCT layer within the UCX framework. In this context, we use the InfiniBand transport layer, although UCT currently fully supports InfiniBand, uGNI, and shared memory interfaces, with comparable performance overheads. At this point, these results are considered preliminary and do not yet fully represent our expected performance overheads.

The evaluation system is equipped with two HP ProLiant DL380p Gen8 servers, each with two Intel Xeon E5-2697 2.7GHz CPUs for a total of 24 CPU cores. The servers are connected to a Mellanox SX6036 switch using a single-port Mellanox Connect-IB FDR host channel adapter (HCA) running firmware version 10.10.5056 and Mellanox OFED 2.4-1.0.4. In addition, we used a prototype implementation



(a) Latency of UCT *short/bcopy*-put and *bcopy*-get operations

(b) Bandwidth of UCT put and get zcopy operations

(c) Message Rate of the UCT short and bcopy operation

Fig. 2: UCX operations using VERBS RC (VERBS) and Accelerated VERBS RC (AVERBS) interfaces

of the Accelerated Verbs driver developed in the context of the Open Fabrics Alliance (OFA) Verbs working group.

To measure the message rate, the bandwidth, and the latency of various operations, we implemented a set of UCT micro-benchmarks. The latency of PUT operations is measured using the ping-pong communication pattern where the initiator of the communication sends a *short* or *bcopy* PUT request from one node to another. Then, the target of the PUT operation replies with a response message. The latency is then the average of half of the round-trip time. The latency of the GET operation is measured using a one-sided benchmark where the initiator reads memory from the target process and reports the average of a full round-trip time. From Fig. 2a we can see that the UCT *short* operation achieves $0.89\mu s$ latency for a one byte message using Accelerated Verbs and $0.91\mu s$ latency with regular the Verbs driver. The UCT *bcopy* latencies are higher due to an overhead imposed by a *memcpy* operation, used for the message transfer to the bouncing buffer. For a one byte GET operation, we observe 1.79 and $1.83\mu s$ latency for the accelerated and regular Verbs, respectively. A similar benchmark is used to measure bandwidth, with the difference that a UCT *zcopy* operation is used; the operations are optimized for bandwidth-bounded operations. Fig. 2b highlights that the *zcopy* operation achieves 6138.5 MB/s bandwidth, which is the maximum practical peak supported at the hardware level for a single FDR (4x) InfiniBand port. Moreover, similar bandwidths can be observed for both one-sided communication, PUT and GET. The message rate is measured by initiating multiple *short* or *bcopy* PUT requests from one node to another. From Fig. 2c we can see that UCT *short* and *bcopy* operations with the Accelerated Verbs driver outperform the same operations on top of the regular Verbs driver by more than a factor of two, and reaches the rate of 14 million operations per second for a single CPU core.

IV. Conclusion and Future Work

UCX is a framework for network APIs, protocols, and implementations. In this paper, we have described the design and architecture of the framework, the interfaces, and protocols for implementing parallel programming models. Further, we evaluated our design decisions by providing a prototype implementation, and analyzing the performance characteristics of important basic primitives required for implementing MPI and OpenSHMEM. Our future work will initially focus on implementing MPI and OpenSHMEM using proposed interfaces and protocols, and later extend UCX to accommodate hybrid programming models and support I/O applications.

V. Acknowledgments

We want to thank Stephen Poole, a co-founder of this project, who helped us with countless hours of technical discussions that made this project a reality. We also want to thank the UH, including Tony Curtis, Pengfei Hao, and Aaron Welch for their feedback.

References

[1] D. Bonachea, "GASNet Specification, v1.1," Berkeley, CA, USA, Tech. Rep., 2002.

[2] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Libray for Ditributed Array Libraries and Compiler Run-Time Systems," in *Proceedings of the 11 IPPS/SPDP'99 Workshops*, UK, 1999.

[3] S. Atchley, D. Dillow *et al.*, "The Common Communication Interface (CCI)," *HOTI*, 2011.

[4] M. ten Bruggencate and D. Roweth, "DMAPP-An API for One-sided Program Models on Baker Systems," in *CUG Conference*, 2010.

[5] R. Brightwell, T. Hudson *et al.*, "Portals 3.3 on the Sandia/Cray Red Storm System."

[6] B. Chapman, T. Curtis *et al.*, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10, New York, NY, USA, 2010.

[7] DOE, "Titan: A Cray XK System at the ORNL," https://www.olcf.ornl.gov/titan/, 2014.

[8] ——, "Summit: IBM's OpenPOWER-based Data Centric System at the ORNL," https://www.olcf.ornl.gov/summit/, 2015.

[9] P. Shamis, M. G. Venkata *et al.*, "Universal Common Communication Substrate (UCCS) Specification. Version 0.1," Oak Ridge National Laboratory (ORNL), Tech Report ORNL/TM-2012/339, 2012.

[10] N. Corp. NVIDIA GPUDirect RDMA. https://developer.nvidia.com.

[11] D. Rossetti, "GPUDIRECT: Integrating the GPU with a Network Interface," in *GPU Technology Conference*, 2015.

[12] Sreeram Potluri, "TOC-centric Communication: A Case Study with NVSHMEM," in *OpenSHMEM User Group Meeting*, 2014.

[13] E. Gabriel, G. E. Fagg *et al.*, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.