OpenSHMEM Reference Implementation using UCCS-uGNI Transport Layer

Tomislav Janjusic, Pavel Shamis, Manjunath Gorentla Venkata, and Stephen W. Poole
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831-6173
{janjusict,shamisp,manjugv,spoole}@ornl.gov

ABSTRACT

OpenSHMEM is a library interface implementation and specification that enables the implementation of the Partitioned Global Address Space (PGAS) model. It exports modern RDMA network functionality and communication semantics to applications very efficiently. There are many closed source implementations of OpenSHMEM for modern RDMA interconnects such as InfiniBand and Cray's Gemini and Aries. Given the important role that Cray systems play in HPC, in this paper, we present an open source implementation of OpenSHMEM for Cray XE/XK/XC systems.

To implement OpenSHMEM, we use the uGNI interface. uGNI is a generic interface that is designed for multiple programming models. The interface fits well the goal of UCCS. Having OpenSHMEM with UCCS-uGNI allows usage of the same implementation over multiple interconnects. This also translates into many advantages that come with common code such as resource sharing, increasing productivity because of less code maintenance, etc. Preliminary results show that OpenSHMEM-UCCS performs comparable to state-of-the-art Cray SHMEM for Put, Get, and AMO operations.

Categories and Subject Descriptors

C.2.6 [**COMPUTER-COMMUNICATION NETWORKS**]: Internetworking—*Standards*

1. INTRODUCTION

OpenSHMEM is a de-facto standard that enables the implementation of the PGAS programming model, and defines the SHMEM library API. It is an open source effort lead by a community including academic, research, and industry institutions. There are many production-grade closed source implementations of the SHMEM library API, SGI's Message Passing Toolkit (MPT), Cray's SHMEM library implementation for Aries and Gemini interconnects, HP's SHMEM library to name a few. Unlike closed source software, the OpenSHMEM open source availability enables users to research and develop new capabilities (extensions) in OpenSHMEM [4].

The current communication middle-ware technologies typically take two approaches. First, a subset of communication packages provide flexible and high performance APIs for specific network hardware technologies; however, this poses a challenge for programming model developers who want to develop codes for multiple platforms. Second, by using a more universal approach and tune APIs for specific programming models they lack broader interface reconciliation. UCCS answers the demand for a universal, low-level, high performance, multi-platform/protocol/network, scalable, and open source network library [5].

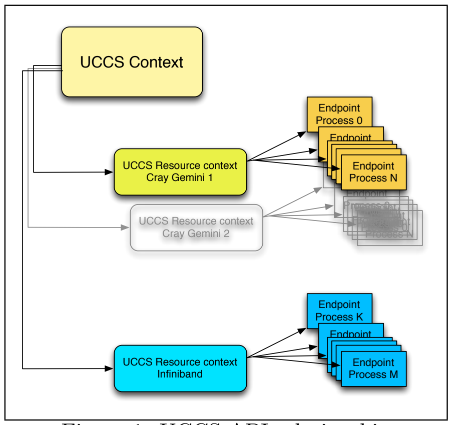
UCCS Context
Endpoint Process 0
Endpoint
UCCS Resource context Cray Gemini 1
Endpoint Process N
UCCS Resource context Cray Gemini 2
Endpoint Process K
Endpoint
UCCS Resource context Infiniband
Endpoint Process M

Figure 1: UCCS API relationship

The Universal Common Communication Substrate (UCCS) is a single low-level communication substrate that exposes high-performance communication primitives, and provides network platform interoperability. The UCCS API is hardware agnostic which isolates the programmer from hardware specific details. It defines several key concepts for achieving data transfers, which are represented using opaque handles that contain all the necessary information. A *communication context* is a method which provides a communication scope and isolates multiple instances of user or system code. A *communication resource* represents a communication channel such as Cray's Gemini, Mellanox' Infiniband, etc. An *endpoint* represents the destination for a communication process. UCCS selects a specific endpoint to complete a communication call. Figure 1 describes the relation between communication contexts, resources, and endpoints. UCCS interface is divided between its core features and the run-time environment (RTE) responsible for providing the necessary run-time services such as process start-up.

The rest of the document is organized as follows. In section 2 we discuss the uGNI interface and current UCCS-uGNI capabilities. In section 3 we discuss are benchmarks and present our preliminary results. In section 4 we present our conclusions and discuss future work. A detailed description of UCCS is available the UCCS' website [3].

2. UGNI OVERVIEW

The user-level generic network interface (uGNI) and distributed memory application (DMAPP) interface provide low-level communication services to user-space software.
uGNI exposes the communications capabilities of the Cray interconnect. The uGNI and DMAPP APIs allow development of portable system software while maximizing the hardware performance of Cray's network interconnect [2]. UGNI is more generic interface enabling implementation of programming models other than PGAS, which is the primary design motivation of the DMAPP interface. For this reason uGNI can be used to extend UCCS' component capabilities to implement other programming models, eg. an MPI programming model. The primary reason is that DMAPP is specifically designed for a one-sided communication model, thus it would require implementing an additional software layer to support message passing which would likely cause additional performance overhead.

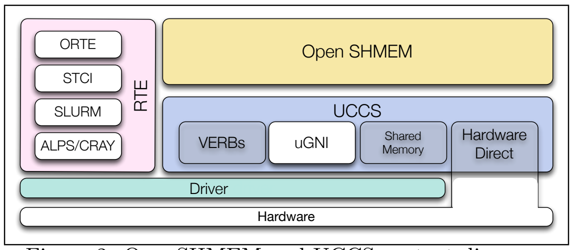

Figure 2: OpenSHMEM and UCCS context diagram.

Figure 2 depicts the relationship between OpenSHMEM reference implementation and UCCS. Moreover, Figure 2 highlights the context of the uGNI component implementation. The UCCS uGNI transport layer supports *Put*, *Get*, and *64bit atomic (AMO)* operations. We discuss Put, Get, and AMO operations performance in Section 3.

The *put* and *get* operations use the *PostFMA* and *PostRDMA* message protocols. Our implementation uses *FMA* protocol for 64K byte or smaller messages, and *RDMA* for larger messages. Due to alignment restrictions, *get* operations perform additional checks in order to fragment the message appropriately.

3. RESULTS

The preliminary results of this implementation were conducted on a Cray system located at Oak Ridge National Laboratory. The testbed system's architecture resembles the Titan HPC system. Each node consists of two 8-core 2.2Ghz AMD Opteron (Interlagos) processor and 32GB of RAM running Cray Linux Environment version . Two nodes share a Cray Gemini high-speed interconnect router. Our tests use the OpenSHMEM reference implementation and the most recent UCCS release. We compare our results against state-of-the-art Cray/SHMEM v7.0.0 library. For our benchmarks we used the Ohio Micro-Benchmark suite [1]. The tests consisted of Put, Get, and 64 bit AMO operations to validate and compare our results.

3.0.1 *Message Put*

The Put benchmark measures the latency of a *shmem_put()* operation for different data transfers. The user selects if the buffer is allocated in global or heap memory. This test performs on exactly 2 processing elements (PEs). PE-1 issues a *shmem_putmem()* operations and then calls SHMEM *shmem_quiet()*. This test is repeated using 1 to 8M byte data transfer sizes and taking the average round-trip latency required to complete each communication call. In this, as well as the following tests, we used the symmetric heap option.

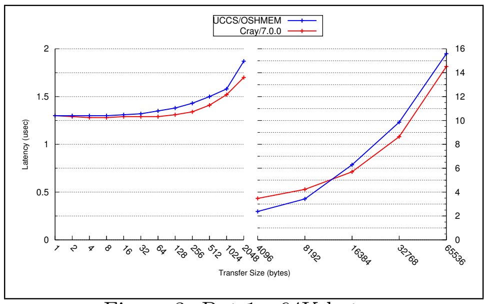

Figure 3: Put 1 - 64K bytes.

In Figure 3 we can observe the latency difference when sending 1 to 64K byte data transfers between PE-0 and PE-1. The results are encouraging showing comparable, $\approx 3\%$ variation, latency results against Cray's SHMEM implementation. The performance switch around 4K byte data transfers can be explained by Cray SHMEM switch from FMA to BTE at 4K byte transfer sizes.

In Figure 4 we can observe the latency difference when sending larger, 128K to 8M byte, data transfers. Similarly, our experiment shows that our put latency is comparable to Cray's SHMEM implementation.

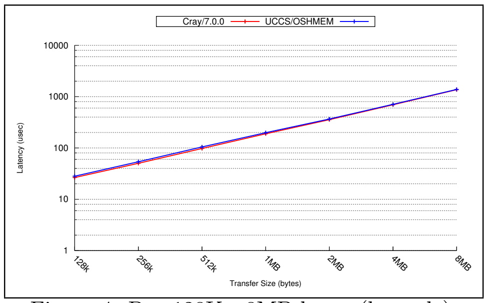

Figure 4: Put 128K - 8MB bytes (logscale).

3.0.2 *Message Get*

The Get benchmark is similar to Put except that the sender, PE-0, issues a *shmem_getmem()* function call to read from PE-1. Similarly to the Put test, the benchmark reports the average round-trip latency required to complete a single Get operation. Figure 5 shows the latency difference for 1 to 64K byte data transfers. In Figure 6 we can observe the latency difference for larger, 128K to 8M byte, data transfers. Due to the relative subpar performance of Cray's *shmem_getmem()* operation Cray recommends to use *shmem_get64* which is tailored for 8byte alligned messages and does not support non-aligned operations. In order to demonstrate performance we modified the original benchmark. Notice the latency disparity manifesting for data transfers larger than 512 bytes using Cray's *shmem_getmem()* operation, and for data transfers larger than 4K bytes using UCCS' *shmem_getmem()* compared to Cray's *shmem_get64()* operation. We believe that there is an optimization issue when using Cray's *shmem_getmem()* operation. Moreover, it is evident from the results in Figure 6 that Cray's *shmem_get64()* operation outperforms ours by a significant margin. We omitted UCCS' *shmem_get64()* results because the underlying implementation does not differentiate between *shmem_getmem()* and *shmem_get64()*.

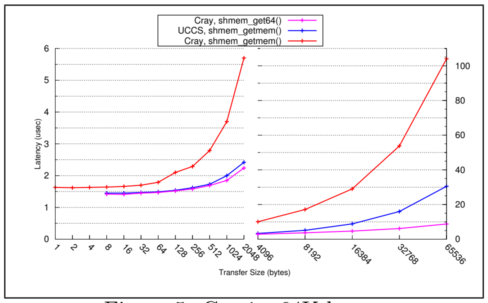

Figure 5: Get 1 - 64K bytes.

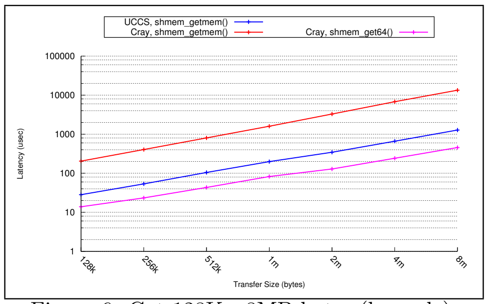

Figure 6: Get 128K - 8MB bytes (logscale).

We are unsure of the exact causes behind our performance degradation and continue to investigate. Our implementation selects code-paths naively based on data transfer size. For data transfers smaller than or equal to 64K bytes we use uGNI's *postFMA* operation, and for larger transfers we use a less optimal *postRDMA* protocol. It is evident from Figure 6 that we must revisit this policy in order to stay comparable to Cray's *shmem_get64()* operation. In future version we will include a special optimization for 8 byte aligned transfers. Overall, UCCS' *shmem_getmem* operation outperforms Cray's *shmem_getmem()*.

3.0.3 Atomics 64bit

The atomic test performs atomic operations and reports a base-line performance for each operation. The benchmark performs a warm-up phase and a measurement phase. The measurement phase reports the average rate and latency of an atomic operation. Figure 8 and Figure 7 show the latency and number of operations per second results for atomic operations, respectively. The swap operation is not natively supported by Gemini ASIC. From Figure 8 and 7 we can observe that UCCS-uGNI outperforms Cray's in *shmem_llong_add* and *shmem_llong_inc()* operations while staying comparable for other operations.

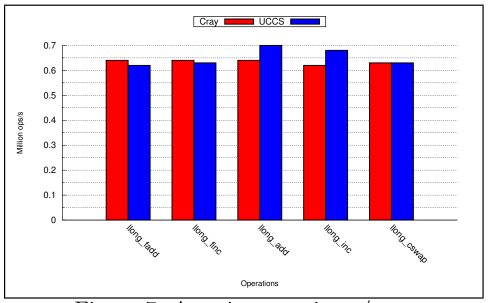

Figure 7: Atomic operations / sec.

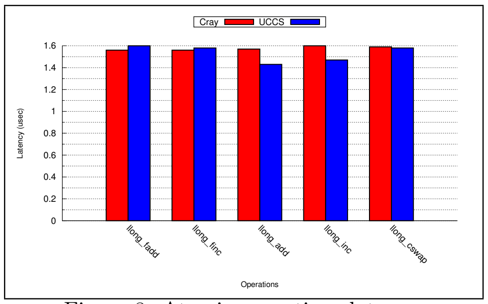

Figure 8: Atomic operations latency.

4. CONCLUSIONS AND FUTURE WORK

The motivation of this paper is to introduce the High-Performance OpenSHMEM reference implementation using UCCS-uGNI component. As part of this effort we identified vendor SHMEM implementation variations which we concluded were likely implementation errors and we notified the vendor accordingly. The UCCS-uGNI transport layer code-paths have to be carefully considered in order to stay competitive. We compared OpenSHMEM reference implentation using UCCS-uGNI transport layer against state-of-the-art Cray SHMEM library. While our initial results are encouraging, there are several issues left to be resolved. First, we must reconsider the Get (*shmem_memget()*) operation to stay comparable with Cray's *shmem_get64()* operation tailored for 8byte aligned data transfers. At present our implementation does not support 32bit and swap AMO operations and we plan to implement them.

Acknowledgment

This work is supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center located at the Oak Ridge National Laboratory.

5. REFERENCES

[1] OSU Micro-benchmark. http://mvapich.cse.ohio-state.edu/benchmarks, 2014.

[2] Cray. Using the GNI and DMAPP APIs. http://docs.cray.com/books/S-2446-3103/S-2446-3103.pdf, February 2014.

[3] ORNL. UCCS website. http://uccs.github.io/uccs, 2014.

[4] P. Shamis, M. G. Venkata, S. W. Poole, A. Welch, and T. Curtis. Designing a High Performance OpenSHMEM Implementation Using Universal Common Communication Substrate as a Communication Middleware. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - First Workshop, OpenSHMEM 2014, Annapolis, MD, USA, March 4-6, 2014. Proceedings*, pages 1–13, 2014.

[5] Shamis, P., Venkata, M.G., Kuehn, J.A., Poole, S.W., Graham, R.L. Universal Common Communication Substrate (UCCS) Specification. Technical Report ORNL/TM-2012/339, Oak Ridge National Laboratory (ORNL), 2012.