Implementation and Evaluation of OpenSHMEM Contexts Using OFI Libfabric

Max Grossman[1(✉)], Joseph Doyle[2], James Dinan[3], Howard Pritchard[4], Kayla Seager[3], and Vivek Sarkar[1]

[1] Rice University, Houston, USA
jmg3@rice.edu
[2] Carnegie Mellon University, Pittsburgh, USA
[3] Intel Corporation, Santa Clara, USA
[4] Los Alamos National Laboratory, Los Alamos, USA

Abstract. HPC system and processor architectures are trending toward increasing numbers of cores and tall, narrow memory hierarchies. As a result, programmers have embraced hybrid parallel programming as a means of tuning for such architectures. While popular HPC communication middlewares, such as MPI, allow the use of threads, most fall short of fully-integrating threads with the communication model. The OpenSHMEM contexts proposal promises thread isolation and direct mapping of threads to network resources; however, fully realizing these potentials will be dependent upon support for efficient threaded communication through the underlying layers of the networking stack. In this paper, we explore the mapping of OpenSHMEM contexts to the new OpenFabrics Interfaces (OFI) libfabric communication layer and use the libfabric GNI provider to access the Aries interconnect. We describe the design of our multithreaded OpenSHMEM middleware and evaluate both the programmability and performance impacts of contexts on single- and multi-threaded OpenSHMEM programs. Results indicate that the mapping of contexts to the Aries interconnect through libfabric incurs low overhead and that contexts can provide significant performance improvements to multithreaded OpenSHMEM programs.

1 Introduction

Over the past decade, the degree of parallelism within high performance computing (HPC) system nodes has increased dramatically through the introduction of accelerators, such as general purpose graphics processing units (GPGPUs), and many-core processors, such as the Intel® Xeon Phi™ processor. Such nodes are often able to achieve peak performance and resource efficiency only when programmed using a node-level programming model, such as OpenACC [12] and OpenMP [15]. At the same time, HPC networking interfaces have also been provisioned to handle communication operations for these large numbers of cores. These drastic shifts in node-level architecture have left conventional, networking-centric HPC programming models, such as OpenSHMEM [16] and MPI [10],

M. Gorentla Venkata et al. (Eds.): OpenSHMEM 2017, LNCS 10679, pp. 19–34, 2018.
https://doi.org/10.1007/978-3-319-73814-7_2

scrambling to provide APIs that are thread-safe, use resources efficiently, and are scalable enough to support hundreds of threads per process.

A first step, taken by MPI 2.0 in 2003 and soon to be adopted by OpenSHMEM, is to make existing HPC communication libraries thread safe. While this addresses the first-order need for threads to perform communication, this approach presents significant challenges, as interference between threads within the middleware and within the semantics of the communication model can lead to significant overheads. For example, in OpenSHMEM's unordered communication model, the *fence* and *quiet* operations are used to ensure ordering and remote completion for operations issued by an OpenSHMEM processing element (PE). When PEs are multithreaded, a fence or quiet performed by any thread will affect operations performed by all threads. Thus, a deeper level of threading integration with the communication middleware is needed to provide thread isolation, enable overlap across threads, and achieve more intelligent resource mapping.

In this work, we focus on the OpenSHMEM communication middleware and the proposed contexts extension for threading integration [3]. In this paper, we present an implementation of the proposed OpenSHMEM contexts extension using the OpenFabrics Interfaces libfabric [14] communication layer and use this as a vehicle to evaluate the above requirements. We utilize the libfabric generic networking interface (GNI) provider to interface with the high performance Aries[1] interconnect. We evaluate our implementation's performance using several representative benchmarks and discuss our experiences developing applications with this new interface, commenting on its programmability and usability characteristics.

2 Background

The SHMEM programming model was first created by Cray Research for the Cray (See footnote 1) T3D machine and has subsequently been supported by a number of vendors across many platforms. The OpenSHMEM specification was created in an effort to improve the consistency of the library across implementations and, more importantly, to provide a forum for the user and vendor communities to discuss and adopt extensions to the SHMEM API.

The OpenSHMEM library provides a single program, multiple data (SPMD) execution model in which N instances of the program are executed in parallel.

[1] Other names and brands may be claimed as the property of others.
Intel and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

Listing 1.1. Proposed OpenSHMEM Contexts API, including examples of contexts version of point-to-point routines and interprocess synchronization routines.

```
int shmem_ctx_create(long options, shmem_ctx_t *ctx);
int shmem_ctx_destroy(shmem_ctx_t ctx);
void shmem_ctx_putmem(shmem_ctx_t ctx, void *dest, const void *source,
                      size_t nelems, int pe);
void shmem_ctx_quiet(shmem_ctx_t ctx);
void shmem_sync_all(void);
void shmem_sync(int PE_start, int logPE_stride, int PE_size, long *pSync);
```

Each instance is referred to as a processing element (PE) and is identified by its integer ID in the range from 0 to $N - 1$. PEs exchange information through one-sided *get* (read) and *put* (write) operations that access remotely accessible *symmetric objects*. Symmetric objects are objects that are present at all PEs and they are referenced using the local address to the given object. By default, all objects within the data segment of the application are exposed as symmetric; additional symmetric objects are allocated through OpenSHMEM API routines. OpenSHMEM's communication model is unordered by default. Point-to-point ordering is established through *fence* operations, remote completion is established through *quiet* operations, and global ordering is established through *barrier* operations.

Recently, thread safety extensions have been proposed for OpenSHMEM [17]. These extensions provide a `shmem_init_thread` routine that can be used to initialize the library with thread safety enabled. Several thread safety levels are provided, with the most notable being `SHMEM_THREAD_SINGLE`, which disables thread safety, and `SHMEM_THREAD_MULTIPLE`, which enables full thread safety. The thread safety extension further defines the behavior of the existing API when used by multiple threads within a PE. In this model, all threads are logically part of the same PE and synchronization actions, such as fence, quiet, and barrier, are performed at the level of the PE. Thus, when any thread performs one of these operations, communication operations performed by all threads are affected.

2.1 OpenSHMEM Contexts

Contexts have been proposed as a means of isolating communication streams, isolating threads from each other, and improving the mapping of threads to underlying network resources [3]. The proposed API extension is summarized in Listing 1.1. Contexts introduce a `shmem_ctx_t` object that is passed to communication and synchronization operations. Thus, operations performed on a given context can be treated separately from those performed on a different context, enabling isolation and overlap across contexts. In effect, each context represents a separate ordering and completion environment, enabling the middleware to efficiently map the communication of different contexts to different communication resources (e.g. transmit engines, command interfaces, or rails). While a single PE can utilize multiple contexts, the PE still represents a single

destination (i.e. PE ID) for SHMEM communication operations. Thus, contexts extend the existing $1 : N$ communication model, where each PE can generate one stream of accesses to N targets in the OpenSHMEM global address space, to an $M : N$ model, where each PE can generate M independent streams of accesses to N targets.

Contexts versions of the `shmem_putmem` and `shmem_quiet` routines are shown to illustrate the extension to the point-to-point API. The full proposal adds contexts version of all point-to-point operations, including put, get, quiet, fence, and atomic memory operations (AMOs).

2.2 Libfabric

Libfabric (OFI) is a vendor-neutral, open interface for high-performance networking applications requiring low latency and high message throughput. The interface was designed by the OpenFabrics Alliance (OFA) Interfaces Working Group (OFWIG), with one of the primary goals of this working group being to define a fabric interface that has a tight semantic map to various applications that use it, including PGAS programming models.

The initial libfabric API and internal design of libfabric has been previously described [7]. Libfabric was designed to provide a vendor-neutral client API that is mapped to a set of *providers* that implement the communication interfaces for a particular fabric hardware. In this paper, we take advantage of key libfabric API features, such as the fine grain transmission context support, to enhance performance and scalability. The latest version of the API and documentation are available online [13]. Libfabrics is freely available on Github [14] and is also distributed via the OpenFabrics Enterprise Distribution (OFED).

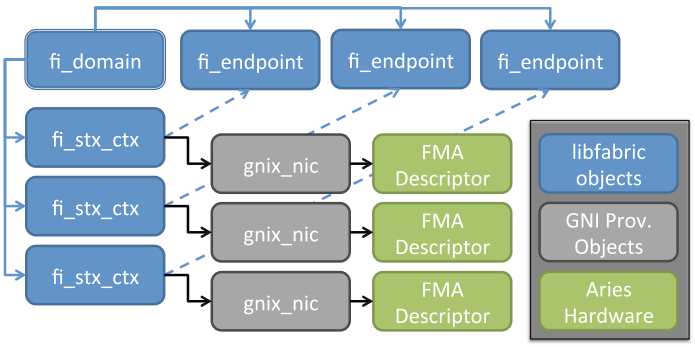


Fig. 1. Relationship of libfabric `fi_domain`, `fi_endpoint`, and `fi_stx_context` objects to GNI provider internal `gnix_nic` objects and underlying Aries hardware. Solid blue lines indicate libfabric objects which are instantiated from a `fi_domain`, while dashed blue lines indicate objects which are associated via an `fi_bind` operation. Black lines indicate associations between libfabric upper level objects and lower-level GNI-provider internal objects and network hardware. (Color figure online)

2.3 Aries and the GNI Libfabric Provider

The Aries interconnect and the GNI libfabric provider have attributes that lend themselves to the investigation of performance gains possible using OpenSHMEM contexts. The Aries interface supports a large number of fast memory access (FMA) descriptors that can be used to enable independent issue of RDMA requests from multiple threads using the same Aries interface. One of the design goals of the GNI provider is to ensure that threads within a multi-threaded process can access the FMA descriptor resources with as little contention as possible [2]. In addition, the Aries interconnect has additional attributes that make it well suited to investigating extensions to the OpenSHMEM API including its ability to offload RDMA transactions and support for an extensive set of 32- and 64-bit atomic memory operations.

For this work, the GNI provider was enhanced to support the libfabric *shared transmission (TX) context* (`fi_stx_context`) construct. The shared context enables multiple endpoints to share an FMA descriptor if transmission resources become scarce. Figure 1 depicts the relationship between libfabric endpoints, shared TX contexts, and the underlying Aries network hardware.

3 Implementation of Contexts over Libfabric

In our previous work, we described an implementation of OpenSHMEM using the OpenFabrics Interfaces libfabric communication layer [19]. This implementation is available as part of the open source Sandia OpenSHMEM (SOS) library [18] and is referred to as the OFI transport layer. The current OFI transport layer was designed to support the single-threaded OpenSHMEM 1.3 programming model; in this work, we extend this layer to support both the proposed thread safety and contexts extensions.

3.1 Middleware Extensions to Support Contexts

The design of the OFI transport layer with threading and contexts support is shown in Fig. 2. The fabric domain represents a handle to the fabric and is the first object created. The OFI transport layer queries libfabric for a domain that can support the required features, including support for the one-sided `FI_RMA` and `FI_ATOMICS` capabilities. Thread safety for libfabric routines is provided by enabling the `FI_THREAD_SAFE` attribute on the fabric domain. Libfabric provides several threading modes; `FI_THREAD_SAFE` was selected because it provides the greatest opportunity for communication parallelism. This mode requests the provider to ensure thread safety, providing the greatest opportunity for fine-grain synchronization at the lower levels of the networking stack. Any other libfabric threading mode would have required SOS to protect calls to the libfabric API with additional locks. Thread safety for internal state in the SOS middleware was implemented using POSIX (See footnote 1) mutexes and separate mutexes are used for each context. Synchronization overheads can be

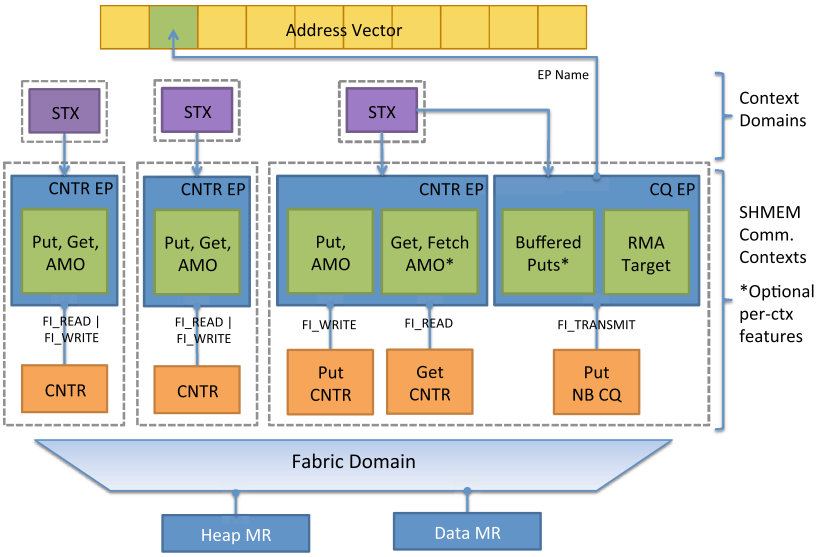


Fig. 2. Architecture of the multithreaded OFI transport layer with contexts support.

further reduced by replacing mutexes with atomic operations, and we plan to investigate this as part of future performance tuning.

From the domain, fabric endpoints (EPs) are created. Endpoints can be used for sending and receiving messages, and the corresponding completion events can be captured as full events in a completion queue (CQ) or as lightweight counting events in an event counter (CNTR). The heap and data segments are registered on the domain and are exposed for remote access through the CQ EP endpoint. The fabric addresses of the RMA target endpoints are queried and exchanged using the process manager in order to populate the libfabric address vector (AV) to provide efficient and scalable translation between OpenSHMEM PE IDs and fabric addresses. Finally, shareable transmit contexts (STX) are created and bound to the endpoints, enabling them to be used for transmitting messages. The STX is "shareable" in the sense that it can be bound to more than one endpoint.

OFI defines its threading model on a domain basis, which forces all EPs on a domain to conform to the same threading model. This model can be restrictive in cases where multiple EPs are bound to an STX, but the application can guarantee that the EPs are not shared by multiple threads (e.g., by setting the `SHMEM_CTX_PRIVATE` flag on the corresponding contexts). From this work we have identified this as a potential performance optimization and are investigating the addition of a synchronization hint on the STX to improve the OFI threading model.

We introduce *context domains* as a means for managing the mapping between OpenSHMEM contexts and fabric resources. Context domains contain the set of resources needed to support a context; minimally, a context domain contains an

STX, but it could also be extended to include resources such as bounce buffer pools, CQs, etc. The maximum number of context domains that can be created is bounded by the `tx_ctx_cnt` attribute on the fabric domain. The maximum number of contexts per context domain is bounded by the `max_ep_stx_ctx` limit. In our current implementation, context domains are created as-needed and the mapping to contexts is controlled manually. In future work, we plan to more deeply explore methods for automatic and efficient mapping of context domains to contexts.

In Fig. 2, we show three context domains each mapped to one context. The rightmost context represents the default (i.e., `SHMEM_CTX_DEFAULT`) context. Optimizations, such as splitting get/put counting events and bounce buffering are optional features that can be enabled on a context. For the backwards compatibility, these optimizations are all made available on the default context. For all other contexts, these optimizations are disabled by default to improve resource utilization. Thus, most contexts are implemented as an EP/CNTR pair that is bound to the STX of the corresponding context domain.

4 Results

In this section, we present quantitative performance evaluation of the OpenSHMEM Contexts implementation described in Sect. 3 and qualitative programmability evaluation of the proposed Contexts API.

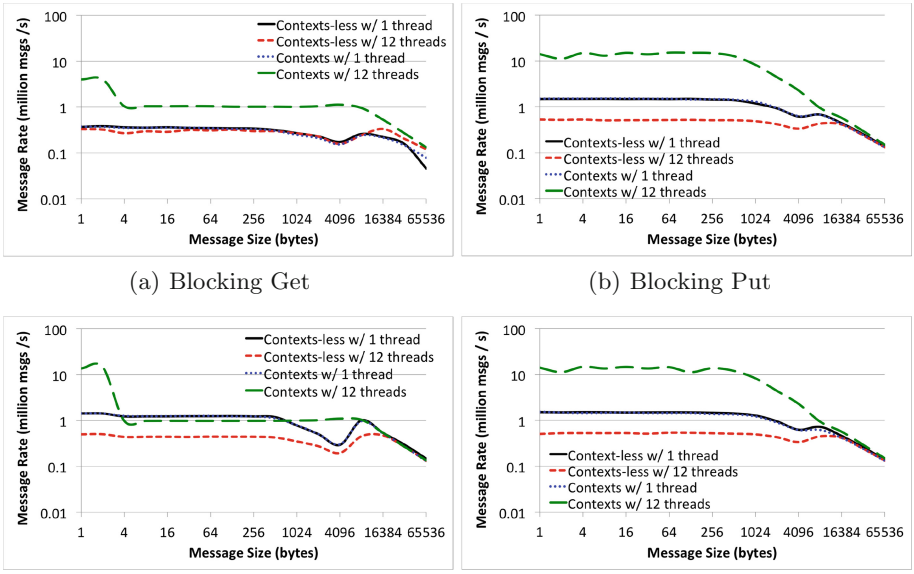
4.1 Evaluation Platform

All experiments presented were collected on the NERSC Edison machine. Edison is a Cray (See footnote 1) XC30 with 2×12-core Intel® Xeon® Processors E5-2695 v2 and 64 GB DDR3 in each node. Edison nodes are connected by the Aries interconnect. All experiments are run on the libfabric-based implementation of Contexts in Sandia OpenSHMEM, as described in Sect. 3. All baseline MPI experiments are run using Cray MPICH 7.4.4. Unless otherwise noted, all tests with hybrid parallelism are run with one PE per socket and 12 threads per PE. All tests with flat parallelism are run with one PE per core.

4.2 Micro-benchmarks

As part of this work, we extended Sandia OpenSHMEM's suite of performance micro-benchmarks to include multi-threaded, contexts-based implementations of all existing micro-benchmarks. For simplicity and platform agnosticism, these multi-threaded benchmarks were written using POSIX (See footnote 1) threads (referred to as pthreads). The benchmarks implemented measure uni-directional and bi-directional bandwidth/message rate for blocking and non-blocking puts and gets. They also measure latencies for blocking and non-blocking puts and gets.

Figure 3 shows the uni-directional put and get rates achieved by contexts-less and contexts-based microbenchmarks using 1 or 12 pthreads. Contexts-less multi-threaded tests rely on libfabric for thread safety. Tests were run using two PEs on two neighboring Edison nodes, with each PE pinned to 12 cores. The rates for non-blocking puts at transfer sizes where an Aries FMA descriptor is utilized show similar improvements to those reported when the libfabric API is used directly [2]. The results obtained for blocking puts are similar owing to a buffering mechanism used by SOS for puts up to 512 bytes. Transfer sizes of 8 KB and higher show little improvement over the single threaded case as these are off-loaded to the Aries RDMA block transfer engine (BTE), which introduces a serialization point.



(a) Blocking Get

(b) Blocking Put

(c) Non-Blocking Get

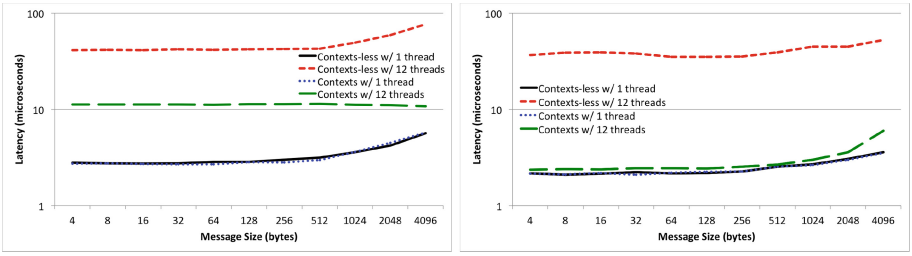
(d) Non-Blocking Put

Fig. 3. Uni-directional get and put rate micro-benchmark results, comparing single-threaded performance with and without contexts with multithreaded performance with contexts. X-axes show message sizes in bytes and Y-axes show achieved message rate in messages per second.

The situation is more complicated for get operations as the target buffer of each get operation must be registered with the Aries device. For very small transfers of 1 to 3 bytes, the GNI libfabric provider uses a mechanism that bypasses its internal memory registration cache and the associated lock. Thus the speedup using contexts for these small transfers is similar to that achieved for put operations. For four byte and larger transfers, the memory registration lock must be taken as the libfabric consumer does not supply a local memory descriptor as part of the call to the libfabric `fi_read` function. To conserve Aries I/O MMU

resource usage, the GNI provider's memory registration cache is associated with a libfabric `domain` object, rather than the underlying `gnix_nic` objects depicted in Fig. 1. The need to acquire this lock significantly limits speedup when using multiple contexts. Note for the single threaded case, performance is significantly better for non-blocking gets owing to the ability to take advantage of pipelining requests to the Aries FMA descriptor.

Figure 4 shows the latencies observed for blocking gets and puts, with and without contexts. In general we observe that using contexts with one thread shows the same latency as not using contexts at all, which is desirable. The put latency remains roughly constant as function of thread count, up to the largest transfer size where bandwidth limitations of the Aries FMA mechanism result in an increase in latency when using 12 threads and contexts. When using 12 threads and no contexts, contention for locks protecting shared endpoints and counter resources leads to large increases in latency.



(a) Blocking Get (b) Blocking Put

Fig. 4. Latency micro-benchmark results, comparing single-threaded performance with and without contexts with multithreaded performance with contexts. X-axes show message size in bytes and Y-axes show latency in microseconds.

These micro-benchmark results illustrate the performance benefits of contexts that previous work [3] has also shown: contexts significantly improve network utilization at small and medium transfer sizes, particularly for put operations. Get transfer rates could be improved either by enhancements to the GNI libfabric provider or by modifying Sandia OpenSHMEM to use pre-registered temporary buffers as the destination for small gets to avoid registration overheads.

4.3 Graph500

Graph500 is an end-to-end benchmark intended to stress the ability of computing systems to support irregular accesses. In this work, we focus specifically on the breadth-first search kernel of the Graph500 benchmark, which traverses a large and randomly generated directed graph. We evaluate two implementations of the BFS kernel using OpenSHMEM Contexts:

1. **Put-Based:** The Put-Based Graph500 implementation partitions vertices of the graph evenly across PEs. A logically global vertex array is partitioned across PEs in their symmetric heaps, where each slot in this global array is an integer that indicates whether the corresponding vertex has been traversed. At each wavefront of the BFS, vertices in the next wavefront are signaled by performing individual puts to the PE that owns those vertices.
2. **Atomics-Based:** The Atomics-Based G500 implementation is similar to Put-Based. However, the global vertex array is instead a bit vector. Signals are sent using atomic bitwise OR operations, rather than puts. This has the benefit of reducing total number of bytes sent, but has the downside of requiring atomic operations.

Figure 5 compares the Put-Based and Atomics-Based G500 implementations against existing OpenSHMEM baselines [6] and reference MPI implementations, performing weak scaling experiments from 8 to 64 Edison nodes. The reference MPI Simple implementation fails to scale to 16 nodes. This is expected, as it is intended to be an example of a readable, easy-to-understand Graph500 implementation, but not a well-performing one. The MPI Replicated implementation, on the other hand, is a hybrid OpenMP+MPI implementation that performs better than all other implementations out to 64 nodes. However, we note that the MPI Replicated implementation is not considered readable; that is, the code structure does not algorithmically reflect the breadth-first search (BFS) operation it implements. Additionally, the scalability of MPI Replicated to larger datasets is questionable. First, it contains a single `MPI_Allreduce` as its only form of computation, hence there is no opportunity for asynchrony or computation-communication overlap. Second, MPI Replicated stores a data structure in each rank whose size scales linearly with the number of vertices in the graph. Larger graphs would hence trigger out-of-memory errors. It is also important to note that as the MPI implementations are run using Cray MPICH 7.4.4, this is not entirely an apples-to-apples comparison.

As described in previous work [6], the OpenSHMEM Checksummed implementation is message-based and to some extent inspired by the MPI Simple implementation. While it outperforms both the Put-Based and Atomics-Based

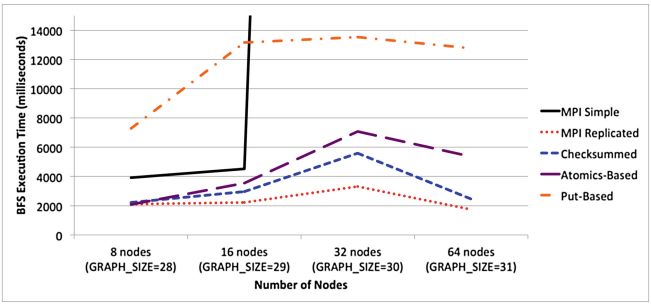


Fig. 5. Graph500 BFS kernel execution time

implementations, it suffers from similar maintainability, readability, and programmability issues as the MPI Replicated implementation.

On the other hand, the code structure of Put-Based and Atomics-Based implementations is much more faithful to the algorithmic structure of BFS. As a result, the RDMAs performed are more fine grain because they model communication along individual edges in the graph. This is exactly the type of communication pattern that generally leads to drastic underutilization of networks. Without OpenSHMEM Contexts, these algorithmically-elegant but generally inefficient communication patterns would not be feasible.

To demonstrate this, we ran two additional experiments. First, we compare running the Atomics-Based implementation with one PE per socket and 12 threads per PE vs. 2 PEs per socket and 6 threads per PE, to see how a reduction in intra-PE parallelism affects performance. We found that using 1 PE per socket ran ∼1.5–2× faster as a result of having more state stored in each PE, reducing redundant cross-PE atomics. Hence, hybrid parallelism enables higher performing Graph500 implementations through the elimination of redundant communication.

Second, we ran the multi-threaded Atomics-Based implementation while relying on runtime-managed thread safety. We keep 1 PE per socket and 12 threads per PE, and compare between giving each thread its own private context vs. simply configuring the runtime as `SHMEM_THREAD_MULTIPLE`. In these tests, we observed that relying on runtime-managed thread-safety led to a slowdown of up to 4×, relative to using contexts.

Therefore, we desire hybrid parallelism in Graph500 and similar benchmarks that exhibit large amounts of small or medium sized communication. However, without contexts, contention between multiple threads within the OpenSHMEM runtime prevents this performance improvement from being realized.

4.4 HPC Challenge Random Access Benchmark

The HPC challenge random access benchmark [9], referred to as the Giga-Updates Per Second (GUPS) benchmark, stresses the ability of a system to perform random memory accesses by performing atomic bitwise XOR updates on a distributed hash table. OpenSHMEM implementations of GUPS that are compliant with the latest ratified specification (OpenSHMEM v1.3) use a get-modify-put pattern to imprecisely emulate bitwise atomic XORs.

In this work, we contribute an OpenSHMEM GUPS implementation based on the APIs in the recently ratified bitwise atomics proposal and show how hybrid programming with OpenSHMEM Contexts improves its performance. We compare two implementations of GUPS: one which is compliant with OpenSHMEM v1.3 and uses a GET, a local bitwise XOR, and a PUT to emulate bitwise atomics, and another which uses the recently ratified bitwise atomics APIs to natively perform the atomic operation.

For both versions of GUPS, we perform three experiments:

1. Flat OpenSHMEM: We run one OpenSHMEM PE per core.
2. Hybrid OpenSHMEM w/Pthreads: We run one OpenSHMEM PE per socket, and run one pthread per core. Each pthread issues atomic updates, gets, or puts (depending on the implementation of GUPS being executed). The OpenSHMEM runtime is configured in thread-safe mode.
3. Hybrid OpenSHMEM w/Pthreads and Contexts: Same as the above, but each pthread has a private context to which it issues OpenSHMEM operations.

Our GUPS experiments are run with a total main table size of ∼500 M words on 16 nodes.

Table 1 demonstrates that the newly ratified bitwise atomics produce a drastic jump in performance, yielding a 2.80× performance improvement when running Flat OpenSHMEM compared to the Gets/Puts-based implementation (rows 1 and 4). When using pthreads without contexts we see the performance cost of purely runtime-managed thread safety with a 14× slowdown when moving from Flat OpenSHMEM with Bitwise Atomics to Hybrid OpenSHMEM with Bitwise Atomics. This slowdown can be attributed primarily to the serialization of accesses to the Aries network AMO hardware when only a single libfabric endpoint is used. However, using contexts allows us to regain that performance plus an additional 1.27× improvement over flat parallelism with bitwise atomics, resulting in a speedup of 3.55× over the baseline.

Table 1. GUPS execution time and speedup relative to the `shmem_long_g/p`-based implementation.

API used	Pthreads used?	Contexts used?	Time (s)	Speedup
Gets/Puts	No	No	60.31	1.00×
Gets/Puts	Yes	No	1042.66	0.06×
Gets/Puts	Yes	Yes	330.08	0.18×
Bitwise atomics	No	No	21.58	2.80×
Bitwise atomics	Yes	No	306.23	0.20×
Bitwise atomics	Yes	Yes	16.97	3.55×

4.5 Mandelbrot

Mandelbrot is a multithreaded benchmark that computes the complex-plane points that are members of the Mandelbrot set. The Mandelbrot implementation used in this paper allows the user to either disable the use of contexts, use contexts for multi-threading, or use contexts in a pipeline. It is also possible to select between blocking and non-blocking OpenSHMEM APIs. We will explore all of these parameters and their effect on performance.

Table 2 shows the performance of Mandelbrot at various runtime configurations, varying whether blocking or non-blocking APIs are used and how contexts

are used in the application. In particular, we note that going from no contexts to using contexts for multi-threading generally yields a ∼2× performance improvement, with pipelining yielding another incremental improvement in performance.

Table 2. Mandelbrot Performance

Blocking APIs?	Contexts?	Execution time (sec)	Speedup
Yes	No	1,285.27	1.00×
Yes	For multithreading	612.13	2.10×
Yes	Pipelined	592.56	2.17×
No	No	1,487.10	0.86×
No	For multithreading	635.34	2.02×
No	Pipelined	608.20	2.11×

4.6 Pipeline Example

We take this benchmark from the OpenSHMEM Contexts proposal itself. This benchmarks demonstrates the use of contexts in a single-threaded C program that performs a summation reduction where the data contained in input arrays on all PEs is reduced into the output arrays on all PEs. The buffers are divided into segments and processing of the segments is pipelined. Contexts are used to overlap an all-to-all exchange of data for segment p with the local reduction of segment $p - 1$.

For these experiments, we perform a reduction on a 16 M element array with staging segments of 4 K elements, using 4 Edison nodes. When running the single-threaded pipeline example without contexts we observed a mean execution time of 7.80 s, versus 6.94 s with contexts (a 12.4% improvement).

4.7 Application Development Discussion

It is important to consider the programmability and usability of any new API being considered for inclusion in the OpenSHMEM specification. While achieving high network utilization is important, the fact is that without usable abstractions it would not be possible to develope large scientific and analytics applications using OpenSHMEM. In our experiences developing these benchmarks on OpenSHMEM we noted several items of interest.

First, contexts enable programmers to write multi-threaded applications with less boilerplate and fine-tuning by making higher network utilization possible at smaller packet sizes. This ability to send data at the natural algorithmic granularity rather than having to manually aggregate and chunk at the application level leads to significant improvement in the clarity of application code.

Second, because contexts facilitate the use of OpenSHMEM in multi-threaded applications, an ancillary benefit is that more state is stored in each PE (assuming some global domain is decomposed across PEs). This low latency, zero copy

access to a larger segment of the data domain often leads to improved performance, even relative to OpenSHMEM implementations that support fast on-node data movement.

Third, in our experience working with these benchmarks we did not find that creating and managing contexts added any significant developer burden. Admittedly, these are not large applications and so our experiences may not apply to development of million-line projects. However, we did not find passing an extra object to communicating routines to be overly burdensome. Additionally, the improvement in composability and network resource partitioning would particularly benefit large applications and libraries.

5 Related Work

The contexts extension to OpenSHMEM was initially proposed in [3], along with an implementation of the API using Sandia OpenSHMEM [18] for the Portals 4 networking layer. They reported performance improvement for two single-threaded OpenSHMEM applications using contexts to avoid unnecessary completion of pending operations and to pipeline OpenSHMEM data transfers. Namashivayam et al. have also presented an implementation of the context extensions for the Aries network using Cray DMAPP as the underlying network API [11].

An alternative approach to OpenSHMEM threading support was proposed by ten Bruggencate et al. [1]. This approach registers thread with the OpenSHMEM runtime in order provide thread isolation and enhance throughput (see [3] for a detailed comparison of contexts and thread registration). Weeks et al. [21] presented results for a set of multi-threaded OpenSHMEM kernels and mini-apps (SHMEM-MT) using these Cray thread safety extensions. Jost et al. [8] have presented a more general discussion of multi-threaded OpenSHMEM applications, including benefits to applications when using a hybrid program model and enhancements to OpenSHMEM to better support such hybrid applications.

Related work in the Message Passing Interface (MPI) community includes the MPI endpoints extension [4,20]. MPI endpoints are conceptually similar to OpenSHMEM contexts in that they allow an MPI implementation to more readily associate individual threads within an MPI process with network resources. Unlike OpenSHMEM contexts however, endpoints are individually addressable so as to allow for sending a message to a particular thread in a target MPI process. There have also been investigations of the performance of MPI-3 RMA operations using multi-threaded MPI applications and micro-benchmarks [5].

6 Conclusion

Contexts extend OpenSHMEM with a programmable abstraction for low-latency, high-throughput access to modern HPC networks. Such programming model primitives are critical as the trend toward multi- and many-core platforms drives applications to utilize hybrid parallelism. This work described a new and

portable implementation targeting the OFI libfabric networking layer. Performance results demonstrated that this API extension maps well to libfabric and can provide significant communication efficiency improvements both for single- and multi-threaded OpenSHMEM applications.

Acknowledgments. This research was funded in part by the United States Department of Defense, and was supported by resources at Los Alamos National Laboratory. This publication has been approved for public, unlimited distribution by Los Alamos National Laboratory, with document number LA-UR-17-26416.

References

1. ten Bruggencate, M., Roweth, D., Oyanagi, S.: Thread-safe SHMEM extensions. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 178–185. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05215-1_13
2. Choi, S.E., Pritchard, H., Shimek, J., Swaro, J., Tiffany, Z., Turrubiates, B.: An implementation of OFI libfabric in support of multithreaded PGAS solutions. In: Proceedings of the 9th International Conference on Parititioned Global Address Space Programming Models, September 2015
3. Dinan, J., Flajslik, M.: Contexts: a mechanism for high throughput communication in OpenSHMEM. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, pp. 10:1–10:9. ACM, New York (2014). http://doi.acm.org/10.1145/2676870.2676872
4. Dinan, J., Grant, R.E., Balaji, P., Goodell, D., Miller, D., Snir, M., Thakur, R.: Enabling communication concurrency through flexible MPI endpoints. Int. J. High Perform. Comput. Appl. **28**(4), 390–405 (2014)
5. Dosanjh, M.G.F., Groves, T., Grant, R.E., Brightwell, R., Bridges, P.G.: RMA-MT: a benchmark suite for assessing MPI multi-threaded RMA performance. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 550–559, May 2016
6. Grossman, M., Pritchard Jr., H.P., Budimlic, Z., Sarkar, V.: Graph 500 on OpenSHMEM: using a practical survey of past work to motivate novel algorithmic developments. Technical report, Los Alamos National Laboratory (LANL) (2016)
7. Grun, P., Hefty, S., Sur, S., Goodell, D., Russell, R., Pritchard, H., Squyres, J.: A brief introduction to the openfabrics interfaces-a new network API for maximizing high performance application efficiency. In: Proceedings of the 23rd Annual Symposium on High-Performance Interconnects, August 2015
8. Jost, G., Hanebutte, U.R., Dinan, J.: Multi-threaded OpenSHMEM: a bad idea? In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, pp. 21:1–21:4. ACM, New York (2014). http://doi.acm.org/10.1145/2676870.2676890
9. Luszczek, P., Dongarra, J.J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., Mccalpin, J., Bailey, D., Takahashi, D.: Introduction to the HPC challenge benchmark suite. Technical report LBNL-57493, Lawrence Berkeley National Laboratory, March 2005
10. MPI Forum: MPI: A message-passing interface standard version 3.1. Technical report, University of Tennessee, Knoxville, June 2015

11. Namashivayam, N., Knaak, D., Cernohous, B., Radcliffe, N., Pagel, M.: An evaluation of thread-safe and contexts-domains features in cray SHMEM. In: Gorentla Venkata, M., Imam, N., Pophale, S., Mintz, T.M. (eds.) OpenSHMEM 2016. LNCS, vol. 10007, pp. 163–180. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50995-2_11
12. OpenACC Standards Committee: OpenACC: Directives for Accelerators (2011). http://www.openacc.org/About_OpenACC
13. OpenFabrics Interfaces Working Group: Libfabric Programmer's Manual. https://ofiwg.github.io/libfabric
14. OpenFabrics Interfaces Working Group: OFIWG libfabric repository. https://github.com/ofiwg/libfabric
15. OpenMP Application Program Interface, Version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf
16. OpenSHMEM application programming interface, version 1.3, February 2016. http://www.openshmem.org
17. OpenSHMEM Redmine Issue #218 - Thread Safety Proposal. http://www.openshmem.org/redmine/issues/218
18. Sandia OpenSHMEM. https://github.com/Sandia-OpenSHMEM/SOS
19. Seager, K., Choi, S.-E., Dinan, J., Pritchard, H., Sur, S.: Design and implementation of OpenSHMEM using OFI on the aries interconnect. In: Gorentla Venkata, M., Imam, N., Pophale, S., Mintz, T.M. (eds.) OpenSHMEM 2016. LNCS, vol. 10007, pp. 97–113. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50995-2_7
20. Sridharan, S., Dinan, J., Kalamkar, D.D.: Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, pp. 487–498. IEEE Press (2014)
21. Weeks, H., Dosanjh, M.G.F., Bridges, P.G., Grant, R.E.: SHMEM-MT: a benchmark suite for assessing multi-threaded SHMEM performance. In: Gorentla Venkata, M., Imam, N., Pophale, S., Mintz, T.M. (eds.) OpenSHMEM 2016. LNCS, vol. 10007, pp. 227–231. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50995-2_16