Enabling Low-Overhead Communication in Multi-threaded OpenSHMEM Applications using Contexts

Wenbin Lu, Tony Curtis, and Barbara Chapman
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, USA
{wenbin.lu, anthony.curtis, barbara.chapman}@stonybrook.edu

Abstract—As the number of shared-memory cores per node in modern High Performance Computing (HPC) machines continues to grow, hybrid programming models like MPI+threads are becoming a preferred choice for scientific applications. While being able to utilize computation resources efficiently, threads in hybrid applications often compete with each other for communication resources, resulting in a negative impact on performance. The OpenSHMEM distributed programming model provides communication context objects that can be used to provide threads with isolated access to the network, thus reducing contention. In this work, we discuss a design for OpenSHMEM contexts and an implementation of the context construct to support hybrid multi-threaded applications and evaluate the performance of the implementation. In all our micro-benchmarks, threads show nearly identical communication performance compared to single-threaded OpenSHMEM processes. By using contexts in hybrid benchmarks, we have achieved up to 43.1% performance improvement for 3D halo exchange, 339% improvement for all-to-all communication, and 35.4% improvement for inter-node load balancing.

Index Terms—PGAS, OpenSHMEM, Hybrid programming, Performance

I. INTRODUCTION

A general trend in High Performance Computing (HPC) hardware is that the machines are evolving towards fewer but more powerful nodes. The Summit supercomputer [1] at the Oak Ridge National Laboratory (ORNL), ranked number one in the TOP500 [2] list as of June 2019, was able to achieve more than eight times the peak FLOPS of the 18,688-node supercomputer TITAN, using only 4,608 nodes. One important contributing factor of this growth is the increase in CPU power, especially in the form of more physical cores per socket, another is the extensive use of accelerators such as GPUs.

To accommodate this paradigm shift in HPC hardware, hybrid programming models, especially MPI+threads, are getting more attention from developers. By utilizing more shared-memory cores per process with fewer processes, hybrid applications are able to exploit extra levels of parallelism, consume less memory, support simple intra-node dynamic load balancing, and reduce synchronization overheads [3].

Despite all the advantages of the hybrid models, poor inter-operability between different programming models has been preventing hybrid applications from reaching the desired level of performance [4]. To maintain thread-safety in most codes, communication calls are either performed outside of the parallel regions by the main thread, or have to rely on coarse-grained locks provided by the runtime of the distributed programming model. Without careful tuning of the hybrid applications, both approaches result in idling cores. The situation is further worsened by the fact that the network adapters used in modern HPC machines usually need to be driven by multiple processors to reach their maximum throughput [5] [6], which is difficult to do for the widely-used one process per socket approach, as processes are usually the only entry points to the network. Last but not least, writing scalable hybrid applications is challenging because the communication model is oblivious of the presence of the threading model, which forces the developers to perform manual message dispatching/tag matching and sophisticated synchronizations [7], reducing the applications’ maintainability. Due to these disadvantages, users of hybrid applications sometimes find themselves searching for the best thread-to-rank ratio for a specific machine through trial and error [8].

Improving the interoperability of different programming models has become an active area of research, with new API/language extensions being proposed and distributed memory runtime libraries receiving threading-related enhancements. The OpenSHMEM [9] distributed memory programming model has included support for multi-threaded communication and synchronization in its latest specification [10].

In this paper, we present a design for, and an efficient implementation of, the context construct in the OpenSHMEM programming model, and evaluate the communication performance improvements in the implementation that lead to high performance in hybrid applications by using benchmarks that resemble the communication patterns used by real-world HPC applications.

The rest of this paper is organized as follows. In section 2, we provide brief introductions to OpenSHMEM, its support for multi-threaded applications, and its communication contexts. In section 3, we show the details of our implementation of the contexts and discuss our design choices. In section 4, we
evaluate the performance of our implementation and discuss the results. Finally, section 5 reviews related work, and section 6 draws conclusions and talks about future work.

II. BACKGROUND

A. The OpenSHMEM Programming Model

OpenSHMEM is a library-based partitioned global address space (PGAS) programming model that allows processes, which are referred to as processing elements (PEs), to exchange data in shared and distributed memory machines. Point-to-point communication in OpenSHMEM is done by blocking or non-blocking one-sided remote memory access (RMA) routines like put, get and atomic operations. However, objects that are allocated on the stack or heap are not accessible by remote PEs. An object that can be modified by OpenSHMEM communication routines is called a symmetric data object, which means there is a corresponding object with the same type and size on all the PEs, residing inside the symmetric heap allocated by the user using OpenSHMEM memory allocation routines, or is a global/static object. Although not its intended purpose, the symmetric heap makes communication in multi-threaded PEs convenient: the user can designate different parts of the symmetric heap to different threads and send data to a thread by simply sending data to that threads private segment on the heap, instead of performing manual dispatch like in MPI’s message tag matching framework. For the example shown in Figure 1, every PE spawns \( N \) threads and allocates a communication buffer \( b[] \) with length \( N \times 100 \) on its symmetric heap. If the programmer assigns each 100-cell segment of the buffer to a different thread, then thread 2 of PE 0 can send a message to thread 1 of PE 3 by sending \( \text{msg}_\text{len} \) cells starting from address \( \&b[200] \) to address \( \&b[100] \) on PE 3, with \( \text{msg}_\text{len} \leq 100 \).

For OpenSHMEM specification version 1.3 and before, no explicit threading support was provided. Hybrid applications had to restrict all communication calls to be performed in the main thread and this limits their flexibility.

B. Threading Support in OpenSHMEM

To provide better support for multi-threaded PEs, version 1.4 of the OpenSHMEM specification \([10]\) allows the programmer to initialize the library with different levels of threading support through the \text{shmem_init_thread} API. Currently four threading levels are defined in the specification (ranked from the most restrictive to least):

- **SHMEM_THREAD_SINGLE** The OpenSHMEM program must not be multi-threaded.
- **SHMEM_THREAD_FUNNELED** Supports multi-threaded PEs but only the main thread (the one that invokes \text{shmem_init_thread}) can make OpenSHMEM calls.
- **SHMEM_THREAD_SERIALIZE** Allows OpenSHMEM calls from different threads in a multi-threaded PE, but the programmer is responsible for making sure that no two threads in the same PE access the OpenSHMEM library at the same time.
- **SHMEM_THREAD_MULTIPLE** Allows any thread to call the OpenSHMEM library at any time, with very few restrictions.

If supported by the OpenSHMEM implementation, specifying the thread level allows the runtime to enable only the necessary thread safety mechanisms for multi-threaded PEs, or disable them and thus eliminate the thread-safety overhead if no threading is needed. In this work, we focus on improving the communication performance of the \text{SHMEM_THREAD_MULTIPLE} mode, as it provides the highest level of flexibility to application developers and its MPI equivalence \text{MPI_THREAD_MULTIPLE} is often criticized for having poor performance \([8]\) \([11]\).

Since specification version 1.4, OpenSHMEM has provided communication contexts \([12]\), with the primary goal of providing independent environments for managing the progression and completion of communication. During its initialization, the OpenSHMEM runtime creates a special context for each PE, denoted by \text{SHMEM_CTX_DEFAULT}, and all communication goes through this context by default. Extra contexts can be created using the \text{shmem_ctx_create} function per user’s request and each one-sided RMA API has a variant that accepts an additional context argument, which this RMA operation will be performed on. A context created with the \text{SHMEM_CTX_PRIVATE} attribute can only be accessed by its creator thread, while creating it with the \text{SHMEM_CTX_SERIALIZE} attribute allows serialized access from multiple threads. A sample usage is shown in Listing 1 with context-related code changes highlighted.

Without the highlighted parts, the code in Listing 1 sends its computation result to the master PE and increases a counter on that PE using atomic fetch-and-add, both going through the default context. To use contexts, the user simply creates two contexts and passes them to the context-variant of the OpenSHMEM routines. When the overlapable computation in \text{foo2} has finished, instead of flushing the default context using \text{shmem_quiet()}, the user should flush the communication using \text{shmem_ctx_quiet}() as \text{ctx}_1 was used to perform the non-blocking put.

At the programming model level, progression and completion of RMA operations performed on different contexts do not affect each other. This allows the programmer to perform fine-
// Create two contexts, both are private to the creator thread
shmem_ctx_t ctx_1, ctx_2;
shmem_ctx_create(SHMEM_CTX_PRIVATE, &ctx_1);
shmem_ctx_create(SHMEM_CTX_PRIVATE, &ctx_2);

// Do some computation, buffer_ptr points to the buffer that stores the result
foo1(buffer_ptr, buffer_len);

// Send the result to the master PE using non-blocking RMA put
shmem_ctx_putmem_nbi(ctx_1, recv_buffer_ptr, buffer_ptr, buffer_len, master_PE_ID);

// Update counter on master PE, and fetch the old value
old_tot_work = shmem_ctx_atomic_fetch_add(ctx_2, &work_counter, buffer_len, master_PE_ID);

// Do more work while data transfer is performed in the background
foo2(...);

// Ensure completion of the RMA put on both sides, buffer can be reused after this call
shmem_ctx_quiet(ctx_1);

// When these contexts are no longer needed
shmem_ctx_destroy(ctx_1);
shmem_ctx_destroy(ctx_2);

Listing 1. Sample usage of OpenSHMEM contexts. Context-related changes from the normal code are highlighted.

gained control of the ordering and/or priority of data transfers. Additionally, if the code in Listing 1 is executed concurrently in a multi-threaded PE, an optimized OpenSHMEM runtime implementation could provide isolated network resources for different contexts, thereby reducing lock contention in high-level abstractions, and utilize multiple network injection points if they are available in the hardware.

III. DESIGN AND IMPLEMENTATION

OpenSHMEM contexts were introduced in 2017 and there are a few implementations. Our group has designed and developed the open-source community reference implementation, called OSSS-UCX [13]. We designed OSSS-UCX to be both our research vehicle for OpenSHMEM extensions, and to be freely available to other developers to use as a basis for their own implementations. One of the main design goals of OSSS-UCX is to enable efficient support of hybrid codes. OSSS-UCX uses UCX [14] as the underlying communication layer. UCX is an open source HPC communication library that abstracts low-level vendor-specific network interfaces, and provides a set of unified communication primitives which can be used to implement various higher-level programming models like MPI and OpenSHMEM. It supports most of the commonly seen HPC interconnects: Infiniband, Omni-Path, CRAY Gemini/Aries, TCP/IP, and shared memory transports. The user-facing UCX protocol (UCP) API hides architectural differences and provides platform agnostic abstractions like RMA operations, tag matching, active messages and data streams.

A. UCP Workers and the Default Communication Context

The UCP worker object is an abstraction that represents a local communication resource and the associated progress engine. Two workers can be connected to generate a UCP endpoint handle, which is used by all UCP communication routines to identify the source and destination of the data transfer. Workers are independent in the sense that the progression of RMA operations in one worker does not affect those of the other workers, even if they are created by the same thread. Tag matching spaces between different workers are also separate, though only relevant to MPI. Implementation-wise, workers are able to select the optimal protocols for different message sizes for each type of interconnect.

In OSSS-UCX, the runtime creates one worker for each PE during the startup process, which we will refer to as the default worker. Then, each PE connects its default worker to all the NUM_PES default workers (including itself) to form a table of NUM_PES endpoints. This default worker and the table of endpoints, together with some other metadata, forms the default context of this PE. When PE N tries to perform some RMA operation to PE M using OpenSHMEM APIs that don’t take a context argument, it picks the Mth entry of the default contexts endpoint table and passes it to the corresponding UCP communication routine to identify the source and the target. This means ordering and progression of communication should also be performed on the default context because we used the progress engine in the default worker. Collective operations and barriers are also performed on the default context.

Although concurrent access to a worker from different threads is supported, and is how some MPI implementations like OpenMPI [15] support MPI_THREAD_MULTIPLE in its UCX backend, UCP workers use coarse-grained locks to maintain thread safety. If user-created contexts simply reuse the same worker in the default context, then all data transfers from different threads in the same PE go through the same worker. For hybrid applications that initialize the Open-
SHMEM library with `SHMEM_THREAD_MULTIPLE`, threads will compete against each other for the lock if they call communication routines at roughly the same time, therefore `SHMEM_THREAD_MULTIPLE` provides no performance advantage when compared to `SHMEM_THREAD_FUNNELED` and `SHMEM_THREAD_SERIALIZED`. This issue is demonstrated in Figure 2 under the `SHMEM_THREAD_MULTIPLE` mode, all the threads spawned by PE \#N compete with each other for communication resources by waiting on a spinlock inside the default context’s UCP worker.

**B. User Created Contexts and Multi-threaded Communications**

Because workers use private communication resources like completion queues and are therefore able to use the finer-grained locks provided at a lower level in the communication library, we choose to map each user-created context to a unique worker. When a new context is created, the runtime will create a UCP worker, and builds its endpoint table by connecting this worker with the default worker of all the PEs. This context can then be passed to context-variants of the one-sided communication routines. When the user destroys a context that is no longer needed using `shmem_ctx_destroy`, OSSS-UCX moves the worker to a worker pool where it can be recycled if the user creates new contexts, as worker creation and wire-up are expensive.

With this implementation, the user can create private contexts for each thread, route one-sided RMA operations through them, and expect performance comparable to single-threaded OpenSHMEM processes. The library-provided collective operations like broadcasts and reductions are still performed on the default context, and must not have more than one thread participating in them from each PE. A diagram of this is shown in Figure 3 where all the threads use their private contexts to do point-to-point RMA operations to avoid lock contentions in the UCP workers.

It should be pointed out that this one-to-one mapping between contexts and UCP workers is essential for certain scenarios: experiments have proved that due to architectural restrictions of UCX, the progression of non-blocking communication will be affected by blocking ones performed on the same worker, and this could make applications miss communication-computation overlapping opportunities unexpectedly. In Listing 1 if contexts are not used or `ctx_1` and `ctx_2` actually shares the same worker, the blocking atomic fetch-and-add will in fact complete the non-blocking putmem before it, and there will not be any communication-computation overlap at all. Our multi-worker implementation ensures the isolation of progression between contexts in similar situations.

**IV. PERFORMANCE EVALUATION AND DISCUSSION**

First, we will evaluate the performance of our context implementation in OSSS-UCX using a set of point-to-point microbenchmarks. These experiments focus on how well the contexts can restore the communication latency/message rate of the threads when compared to single-threaded OpenSHMEM processes. Additionally, we run three advanced benchmarks: each uses a different communication pattern that is commonly seen in real-world HPC applications to evaluate the benefits of our implementation in more realistic situations.

Our experiments were conducted on the SeaWulf/LI-red cluster at Stony Brook University. Nodes used in this study are equipped with two Intel Xeon E5-2690v3 CPUs with 12 cores each (hyper-threading is disabled), 128GB of memory, and one Mellanox Connect-IB FDR (54.54Gb/s) network adapter.

Single-threaded OpenSHMEM benchmarks are initialized using the `shmem_init()` routine, which is equivalent to `SHMEM_THREAD_SINGLE` in OSSS-UCX. Hybrid benchmarks initialize the OpenSHMEM runtime library with the `SHMEM_THREAD_MULTIPLE` mode, unless only one thread is requested, in which case they choose the `SHMEM_THREAD_FUNNELED` mode to reduce threading-related overhead. Threads/Processes are pinned to their dedicated CPU cores to avoid the context switching cost. Our choice of the threading library is OpenMP [16], mainly for its
convenient thread placement control and built-in synchronization constructs. For detailed information on the software stack and run time parameters, please refer to the reproducibility appendix A.

A. Point-to-point Unidirectional Message Rate Benchmarks

We forked the Ohio State University (OSU) OpenSHMEM micro-benchmarks [17] by adding support for benchmarking multi-threaded PEs, and support for contexts. The message rates of three different PE configurations were measured: multi-threaded PEs with all threads in the same PE share the default context; multi-threaded PEs with every thread using its private context; and single-threaded OpenSHMEM PEs. The experiments were performed on two adjacent nodes that are connected to the same Infiniband switch to reduce interference. In the hybrid versions, thread $t$ of PE 0 perform RMA operations on the symmetric heap segment of thread $t$ of PE 1 running on the other node, with no overlap between each thread’s heap segments. For the single-threaded OpenSHMEM version, PE $x$ in the first node perform RMA operations on the symmetric heap of PE $x + 12$, running on the other node, with $x \in [0, 11]$. This setting ensures that the communication patterns of different configurations are identical. Note that for both putmem and getmem, we use as many as 12 threads/processes, as going over the number of physical cores in each socket introduces non-uniform memory access (NUMA) overheads to the measurements, especially for large message sizes.

Figure 4 shows the message rates of the RMA operation putmem and getmem, using power-of-two message sizes from 1 byte to 1 megabyte. Axes in different sub-figures of the same operation are the same.

For putmem, the single-context version suffered from heavy thread contention in small to medium message sizes, with message rate as low as only 5.7% of the single-threaded OpenSHMEM version for message size of 2 bytes under 12 threads. The multi-context version has much better performance, with a single outlier having only 32.7% of the message rate of the single-threaded version for 256-bytes messages. It even outperforms the single-threaded version when only one thread is used, probably due to having a dedicated worker for the RMA operations in that thread.

For getmem, the results are more stable and it is clear that the multi-context version has performance very close to the single-threaded version (the difference never exceeds 9%). When using 12 threads per socket, the single-context version has only about 7% to 10% of the message rate of the single-threaded version in the 1 byte to 1 kilobyte range.

Results for atomic operations are presented in Figure 5. Since the messages are only 4 bytes, NUMA effects are not very noticeable and so we use as many as 24 threads/processes per node. For both types of operations, the multi-context version and the single-threaded OpenSHMEM version have nearly identical message rate, while the single-context version is up to 44% slower for atomic post and 80% slower for atomic fetch.
Results collected through these point-to-point unidirectional micro-benchmarks have shown that threads gained communication performance comparable to single-threaded OpenSHMEM processes, using private communication contexts. However, these results are obtained by maximizing potential contentions in the OpenSHMEM runtime, and the effectiveness of the multi-context configuration needs to be tested with more realistic hybrid applications.

B. Ping-Pong Message Latency Benchmarks

The ping-pong test measures the latency of the round-trip of a message between two processors. We have extended this benchmark to support measuring multi-threaded message latency, following the suggestions in [18]. Again, two adjacent nodes were used in our experiments and processor $n$ in one node only exchanges messages with processor $n$ on the other node. We perform $2^{13}$ warm-up iterations and $2^{20}$ timed iterations of the ping-pong benchmark, and calculated the average iteration time as the round-trip latency. Results are shown in Figure 6.

In both the putmem variant and the atomic fetch variant, the single-context version is much slower at high thread counts, with message latency 8.73 and 2.66 times higher than the single-threaded OpenSHMEM version, respectively. For the atomic fetch variant, the multi-context version outperforms the single-threaded version, the reason is still under investigation.

C. 3D Halo Exchange Benchmark

For scientific simulations that use spatial discretization, the numerical schemes usually have stencils that create data dependencies between several neighboring mesh points. It is common practice for a processor to store duplicated parts of the boundaries of its neighboring processors’ working sets. These are called halo regions and are necessary for processes to avoid fetching mesh data from their neighbors point-by-point with multiple expensive communication routines. In every iteration of the simulation, each processor needs to exchange its halo data with all of its neighbors to keep them up to date. These duplicated halos increase the applications’ memory footprint and the amount of data being exchanged in every iteration. The percentage of extra memory consumption can be calculated with equation

$$\frac{(m + 2h)^d - m^d}{m^d} \times 100\%$$

where $m$ is the size of the mesh in each PE, $h$ is the depth of the halo region in number of mesh points, and $d$ is the dimension of the simulation space. For a 3D simulation using a $500 \times 500 \times 500$ mesh and a 1-point deep halo, a $5 \times 5 \times 5$ process grid will increase the memory usage by 6%, while a $10 \times 10 \times 10$ process grid will raise that number to 12%. This memory overhead grows significantly for larger number of PEs, higher simulation dimensions, and deeper halos.

Shared-memory programming models like OpenMP can be added to reduce the number of working sets while maintaining an efficient utilization of the computing resources. In this hybrid model, it is desirable to also split up the work of packing/unpacking and exchange of the communication buffers to different threads, for better data locality and load balance. Unfortunately, past work has shown that unoptimized threading support in the communication library resulted in thread contention overheads and affected the application’s scalability.

In this benchmark, we perform the halo exchange that would be used for a 3D simulation: for each time step, every PE exchanges halo data with its six neighbors. Instead of distributing an increasing number of processes/threads evenly across a fixed number of nodes, we choose to always use all 24 cores in a node and increase the number of nodes, as this
is how HPC clusters are used in practice. This strategy is used for the rest of the benchmarks in this paper.

The simulation domain has $3072 \times 3072 \times 3072$ mesh points, partitioned to the number of sockets for the hybrid versions, or to the number of cores for the single-threaded OpenSHMEM version. Since they use different halo sizes and the hybrid version only needs up to six cores per socket to do communication, comparison between the single-threaded versions and the hybrid version is inherently unfair. The results of the single-threaded OpenSHMEM version are nevertheless included to demonstrate the performance hit of the thread contention in the single-context version makes it slower than even the naive approach. We perform halo exchange for 500 iterations and take the average iteration time as our result.

In Figure 7, we can see that the hybrid versions have similar behavior when scaling up, and the reduced thread contention in the multi-context version has made it 43.1% faster than the single-context version for running on 32 nodes. The single-threaded OpenSHMEM version does not scale as well, mainly due to the bigger total size of the halo regions. This demonstrates that our multi-worker approach can help hybrid applications take advantage of shared memory programming models without incurring a negative impact on communication performance.

D. Integer Sort Benchmark

All-to-all communication, where every pair of PEs exchange data, is another communication pattern widely used by HPC applications. For exchanging fixed-length data like in Fast Fourier Transform (FFT) and matrix transposition, the OpenSHMEM standard provides the `shmem_alltoall` API. If the sizes of the messages are not uniform across different PEs, the programmer will have to perform the exchange manually because although there was some past work about this kind of usage [19], the equivalent of `MPI_Alltoallv` does not exist in OpenSHMEM yet. This communication pattern is much more intense than the nearest-neighbor exchange pattern used
The ISx integer sort benchmark \cite{20} performs distributed bucket sort on integer keys and it has the all-to-all exchange communication pattern. Since the keys are randomly generated at run time and follows a uniform distribution, the sizes of the messages are slightly different for each pair of PEs, and so the benchmark have to implement its own MPI\_Alltoallv equivalent. In our experiments, we will use a fork of ISx that has support for multi-threaded PEs. During the exchange, each CPU core will send keys from its local buckets to the private receiving buffers of the other cores using non-blocking OpenSHMEM calls in a round-robin fashion, as if they are individual processes. The schedules of the different versions of this benchmark are carefully arranged so that they are equivalent and do not create any communication hot spots. In practice, doing this thread-to-thread exchange may expose more opportunities for computation-communication overlapping as communication can start before all the data are ready \cite{21}. In our experiments we only measure the communication timings, without performing any other operations on the data.

We measured the weak scalability of the ISx benchmark with $10^5$ keys per core, 24 cores per node, and ranging from 2 to 32 nodes. The hybrid versions still use one PE per socket to avoid any NUMA effects. Results shown in Figure 8 are calculated by taking the average time of 100 iterations.

Both the multi-context version and the single-threaded OpenSHMEM version scale better than the single-context version, with the multi-context version runs 339\% faster on 32 nodes/768 cores. This bigger difference in performance is expected since the all-to-all exchange is more intensive and therefore puts higher pressure on the runtime’s thread-safety mechanism. It is not entirely clear why the multi-context version has better scalability than the single-threaded version, especially since all the threads use private key buffers and therefore should have similar cache efficiency when compared to the single-threaded version. One possible explanation is that the single-threaded OpenSHMEM version uses 12 times more PEs, this creates higher overheads in different parts of our software stack and they get magnified by the intensive data exchange.

### E. Mandelbrot Set Benchmark

This benchmark is taken from \cite{12}. It generates a grey scale image of the Mandelbrot set using the quadratic iteration function:

$$z_{n+1} = z_n^2 + c$$

The image is partitioned evenly across the PEs and the computation is embarrassingly parallel. However, the pixels on the image can take very different numbers of iterations to compute, which means that distributed load-balancing is required to reach high computation efficiency.

The work-stealing algorithm used by this benchmark for inter-node load balancing can be described as follows: when a thread finishes a local work set, it queries another PE for unfinished pixels using an atomic fetch-and-add, then it will compute the fetched work set for the victim PE and send the results back, or query the next PE if the previous one has no work left to be done.

Experiments were performed to measure the strong scalability of the three versions. We compute a $3200 \times 3200$ image of the Mandelbrot set with a maximum of 4096 iterations before declaring a point does not belong to the set. The aggregated work rates (points per second) are recorded and presented in Figure 9.

The multi-context version scales slightly worse than the single-threaded OpenSHMEM version, as it was able to achieve 93.9\% of the work rate of the single-threaded version.
on 32 nodes, and is 35.4% faster than the single-context version in the mean time. The unstable performance of the single-context version is caused by the naturally imbalanced load in the Mandelbrot set computation: the way we partition the image changes as we increase the number of nodes, this creates/eliminates computational hot spots. Thread contention in the single-context version hinders the work stealing step and so it shows unstable behavior.

V. RELATED WORK

There have been numerous efforts in the past that have tried to improve the communication performance of hybrid OpenSHMEM applications by providing optimized contexts. In the original OpenSHMEM context extension paper [12], an implementation called Portals-SHMEM [22] was presented, which was built on top of the Portals 4 networking API [23]. In Portals-SHMEM, contexts have their own private memory descriptors and counter objects, which are used by the Portals library to drive communication and keep track of the number of remote completions, respectively. This approach has achieved similar performance improvements in the Mandelbrot set computation benchmark. The single-threaded OpenSHMEM version of the ISx benchmark was also evaluated to show the benefits of communication pipelining. However, the hybrid versions of ISx were not tested.

Based on the libfabric communication framework [24], Sandia OpenSHMEM (SOS) [25] was used in [26] to test similar improvements to contexts. Different contexts in SOS are mapped to different shareable transmit contexts (STX), which in turn are bound to one or more fabric endpoints that belong to the same PE-specific fabric domain. Since the fabric domain provides thread-safety guarantees and the endpoints have independent completion queues, this approach also avoids locking in high-level abstractions. Significant improvements were seen in several benchmarks that have irregular data access patterns.

OpenSHMEM-X [27], developed at ORNL, is another implementation of OpenSHMEM that provides threading-optimized contexts. It is similar to OSSS-UCX in the sense that it also uses UCX as the communication backend, creates multiple UCP workers to reduce thread contention, and has achieved similar speedups in micro-benchmarks. However, OpenSHMEM-X only creates up to one worker for each thread and it relies on OpenMP runtime APIs to identify individual threads. When a thread tries to create more than one private context, all these contexts will reuse the same thread-specific worker OpenSHMEM-X created for its first private context. Shared contexts created by the application are simply aliases of SHMEM_CTX_DEFAULT. This approach suffers from the issue of early completion of non-blocking RMA operations mentioned near the end of section [11-B] and makes OpenMP a dependency.

Cray SHMEM [28] also experimented with different ways to improve their support for hybrid applications. In their context implementation, each context can have a unique Communication Domain (CD), which has its own Completion Queue (CQ) and other network resources. The Thread-Safe extension to OpenSHMEM, proposed by Cray, was also evaluated. In Thread-Safe, any thread that calls OpenSHMEM routines must first register itself via shmem_thread_register, and this allows the library to map threads to network resources like CDs and CQs. The Cray paper focused on comparing the performance of different proposals and different implementation details.

Extensions similar to OpenSHMEM contexts were also proposed for MPI. The MPI endpoints [7] extension is more advanced than OpenSHMEM contexts in the sense that it allows hybrid applications to create new communicators among a subset of the threads they spawn, so the participating threads can have unique ranks and perform collective operations as if they are MPI processes. A library based implementation of MPI endpoints was presented in [11], in which communication calls made by threads with the endpoint communicators are redirected to hidden proxy MPI processes to achieve speedups similar to our work. More recently, MPI finepoints [29] was proposed to improve the communication performance of multi-threaded MPI applications. By allowing threads/tasks to participate in different partitions of the same send/recv operation, finepoints can reduce synchronization cost, eliminate tag matching overhead, perform aggregation of small messages, and exploit other features of modern network adapters. The finepoints implementation has shown good improvements for both micro-benchmarks and mini-applications, but the API is not as flexible as MPI endpoints or OpenSHMEM contexts.

VI. CONCLUSIONS AND FUTURE WORK

As we march toward the era of exascale computing, it is crucial to address the discrepancy between the ever growing on-node parallelism and the decade-old design decisions baked into our software stack, through the development of efficient interfaces between different programming models. This paper has presented a design and implementation of contexts in the OpenSHMEM programming model, and demonstrated its benefits for improving the communication performance of the threads in hybrid applications. By implementing a one-to-one mapping between user-allocated contexts and UCP workers, threads can take advantage of independent message queues, instead of funneling all communication through the same entry point to the network. Essentially, this approach replaces the coarser-grained locks with finer-grained ones that reside in lower levels of the communication layer, thereby delaying thread-safety protections until necessary.

Synthetic micro-benchmarks show that for various RMA operations, threads in the multi-context configuration have performance very close to the reference performance obtained through a single-threaded OpenSHMEM version of the benchmarks, and are consistently faster than the threads in the single-context configuration, especially for small message sizes. For communication patterns that are commonly found in scientific HPC applications, the multi-context version has achieved a speedup of 43.1% for halo exchange, 339% for all-to-all exchange, and 35.4% for work-stealing dynamic load...
balancing. This shows that through our implementation of contexts, hybrid OpenSHMEM applications are able to harness the power of shared-memory programming models without suffering from degraded communication performance.

For future work, we plan to explore how contexts can benefit applications that use task dependencies/priority to express sophisticated workflows. For these applications, contexts could be used to realize communication priority and reduce the interference between tasks due to contention in the communication library. Another possibility is to see how our implementation performs under light-weight user-level threads like Argobots [30], especially since it may not be profitable to allocate one UCP worker for each user-level thread, and we may need to use the shared contexts created by specifying $\text{SHMEM\_CTX\_SERIALIZED}$ during their creation. Lastly, adding support for user-created contexts to collective operations could further reduce the interference between threads and improve applications’ performance.

ACKNOWLEDGMENT
This research was funded in part by the United States Department of Defense, and was supported by resources at Los Alamos National Laboratory, operated by Triad National Security, LLC under Contract No. 89232318CNA000001. The authors would also like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the high-performance SeaWulf computing system, which was made possible by $1.4M National Science Foundation grant (#1531492).

REFERENCES
APPENDIX

Artifact Description: Enabling Low-Overhead Communication in Multi-threaded OpenSHMEM Applications using Contexts

A. Abstract

This artifact description describes the dependencies and experimental setup used to generate the performance results presented in this work.

B. Description

1) Check-list:
   - Program: C++, MATLAB
   - Compilation: GNU C++14 compiler
   - Run-time environment: CentOS Linux 7.5 x86 64
   - Hardware: SeaWulf HPC cluster at the Stony Brook University, 24-core nodes
   - Execution: Parallel
   - Output: Elapsed running time stored in text files
   - Publicly available?: Yes

2) How software can be obtained (if available): All benchmarks, job scripts, results and MATLAB plotting scripts are freely available as GitHub repository: https://github.com/lyu/paw19

3) Hardware dependencies:
   - Processor: Two 12-core Intel Xeon E5-2690 v3
   - Memory: 128GB DDR4
   - Network adapter: Mellanox Connect-IB MT27600

4) Software dependencies:
   - GNU gcc & g++ 9.1.0
   - MLNX_OFED libibverbs 4.4.2.0.1.44207
   - UCX: GitHub master branch, commit acf0152
   - hwloc 2.0.4
   - PMIx 3.1.3
   - PRRTE 1.0.0
   - OSSS-UCX: GitHub master branch, commit 690f54d
   - MATLAB R2019a (Only for generating plots)
   - Torque 6.1.1

5) Data sets: N/A

C. Installation

The configuration and installation of OSSS-UCX and its dependencies are described in install_osss.sh in the GitHub repository. This may require modifications for a different run-time environment.

D. Experiment workflow

The experiments are performed by allocating 32 nodes using the Torque resource manager, and running all the four benchmark scripts in that allocation. The results are printed to stdout and redirected to log files.

E. Notes

To minimize the interference of communication performance from jobs running in neighboring nodes, we allocated 32 nodes that are relatively close to each other. All benchmarks were run multiple times to make sure we are getting consistent results.

For hybrid versions of the benchmark, different OpenMP threads are bound to different cores using the environment variable OMP_PROC_BIND=true to reduce the cost of context switch. The single-threaded OpenSHMEM version was launched with osrun options --rank-by core:span --bind-to core, so that the processes have similar ranking and binding with threads in the hybrid versions.

Run times of the benchmarks are collected through high-precision monotonic timers from inside the application. Time spent on initializing the library, preparing the message buffers and allocating OpenSHMEM contexts was excluded from the total elapsed running time.

In the hybrid versions of some of the benchmarks, thread contention could result in very different iteration times in different threads. For each iteration, we do not stop the timer until the last thread has arrived at the OpenMP barrier, and we take the arithmetic mean of the iteration time as the final result.