Contexts: A Mechanism for High Throughput Communication in OpenSHMEM

James Dinan and Mario Flajslik
Intel Corporation

ABSTRACT

This paper introduces a proposed extension to the OpenSHMEM parallel programming model, called communication contexts. Contexts introduce a new construct that allows a programmer to generate independent streams of communication operations. In hybrid executions where multiple threads execute within an OpenSHMEM process, contexts eliminate interference between threads, and enable the OpenSHMEM library to map operations generated by threads to private communication resource sets. By providing thread isolation, contexts eliminate synchronization overheads and enable each thread to drive a similar set of resources and achieve performance comparable to an OpenSHMEM process. In conventional, single-threaded execution, contexts provide greater control over ordering of operations and can improve communication and computation overlap. A detailed description of the contexts interface and its implementation for the Portals 4 network programming interface is described. The implementation is evaluated using Mandelbrot set and integer sorting (IS) benchmarks. Contexts provide a 25% performance improvement for Mandelbrot by eliminating thread interference and enabling pipelining, and a 35% improvement was achieved for IS by enabling more effective communication/computation overlap.

1. INTRODUCTION

Current trends in high performance computing (HPC) systems indicate that the number of cores per node will continue to increase, and that there will be a commensurate increase in the capabilities of the system interconnect to support high rates of communication through all cores. Hybrid parallel programming models that combines a system-level communication library, such as OpenSHMEM [12] or MPI [9], with a node-level, shared memory parallel programming model have become increasingly prevalent as a means to align applications with systems and enable more efficient use of node-level resources. In such scenarios, a single instance of the communication library is shared by many threads, requiring synchronization within the library and mapping the operations of all threads to a shared set of communication resources.

HPC networks also allow increasing degrees of out-of-order message delivery, for example to route around congestion, providing more efficient use of the network, but introducing additional complexity in the management of communication operations. While such architectures are compatible with existing communication libraries, not all parallel programming models are well-equipped to express independence between communication operations, which is required for efficient use of such networks. For such models, this artificial dependence can limit opportunities for overlapping communication with computation.

OpenSHMEM is a one-sided communication library that provides a partitioned global address space (PGAS) parallel programming model. The OpenSHMEM specification represents an ongoing community effort to standardize the SHMEM parallel programming interface that has been in use for over two decades. During this process, the community seeks to introduce new features to enhance the performance and capabilities of SHMEM on current and future HPC systems. A key activity in this effort is defining the interaction of OpenSHMEM with threads and ensuring that OpenSHMEM will achieve high performance in multithreaded environments.

In this paper, we present OpenSHMEM contexts, a proposed extension to the OpenSHMEM interface that is intended to address these challenges. Contexts introduce separate communication streams that can be isolated to enable efficient communication in hybrid SHMEM-and-threads executions and on out-of-order networks. In particular, contexts address the following performance challenges to both multithreaded and single-threaded SHMEM programs: (1) thread interference, by allowing threads to generate independent streams of communication operations; (2) multithreaded communication throughput, by enabling a mapping of a thread's communication operations to independent network injection resources; (3) out-of-order communication, by providing a mechanism to express independent streams of communication operations; and (4) overlap of communication with computation, by providing greater control over completion of nonblocking operations through completion of individual streams.

We present an implementation of OpenSHMEM contexts in the open source Portals-SHMEM library [13]. Portals-SHMEM utilizes the Portals 4 [16] low-level network API, which is representative of a modern, high-performance, offload network. We analyze the effectiveness of the contexts interface and evaluate its performance using a multithreaded Mandelbrot set benchmark and a single-threaded Integer Sort (IS) benchmark from the NAS parallel benchmark suite. Through the addition of contexts, we observed a 25% performance improvement for Mandelbrot by eliminating thread interference and enabling pipelining, and a 35% performance improvement for IS by enabling more effective overlap

of communication and computation. Results indicate that contexts provide isolation across communication streams, yielding significant improvements in multithreaded communication, as well as significantly improving overlap in single-threaded communication.

The rest of this paper is organized as follows. In Section 2 we provide background on the OpenSHMEM parallel programming model. In Section 3 we present the OpenSHMEM contexts extension and we describe its implementation on top of Portals 4 in Section 4. We present an experimental evaluation in Section 5, discuss design alternatives and tradeoffs in Section 6, and compare our approach with related work in Section 7. We conclude with Section 8.

2. BACKGROUND

SHMEM is an SPMD library that enables multiple processes, referred to as processing elements (PEs), to exchange data through one-sided *get* and *put* memory copy operations, as well as one-sided atomic operations. A point-to-point ordering in the local PE's communication operations can be induced through the *fence* operation. A stronger operation, *quiet*, ensures remote completion and global visibility of all operations issued by the local PE. In addition, the quiet operation also completes all outstanding nonblocking operations. Data in SHMEM PEs is private by default, and data is shared through symmetric objects where an instance of the object is allocated at every PE. Dynamically allocated symmetric objects are stored in a symmetric heap and all static objects in a SHMEM program's data segment are shared automatically through a symmetric data segment.

The SHMEM parallel programming model was developed in 1994 [2, 5], as a high performance parallel programming interface for the Cray T3D family of systems. Over the subsequent years, SHMEM has been supported by a number of vendors, and many have generalized and expanded upon the original interface.

Recently, OpenSHMEM [12], a community-driven open standard, has emerged as an effort to standardize SHMEM and extend its interface to better serve users. Among the goals of the OpenSHMEM effort are enabling efficient hybrid parallel programming that combines OpenSHMEM with a shared memory programming model (e.g. OpenMP), and enabling better overlap of computation with communication [15]. For the remainder of this paper, we use the names OpenSHMEM and SHMEM interchangeably.

2.1 Portals 4 Network Programming Interface

In this work, we demonstrate the communication contexts extension using the open source OpenSHMEM implementation for the low-level Portals networking API [3, 13]. The Portals interface exposes sections of a process' address space for one-sided remote access using read, write, and atomic operations. Accesses to exposed memory regions can be guarded through matching criteria that are used when implementing matched, or two-sided, communication operations. For one-sided communication, a non-matching interface is provided that allows all operations targeting the process to access the given memory region.

The ordering of operations is an important component in synchronization for one-sided communication models. Portals presents the programmer with an unordered network model, where data is not guaranteed to arrive at the target in the order in which it was sent. This delivery model enables dynamic message routing, and also ensures reliable delivery. When a message has been delivered to the target, an acknowledgement message is returned to the sender. Thus, when a process waits for communication operations to complete, it waits for acknowledgement messages from the target. Acknowledgements can generate a full event that includes detailed information about the data transfer, but they are most commonly aggregated into a counting event, which generates less overhead and counts the number of acknowledgements that have been received.

3. COMMUNICATION CONTEXTS

The OpenSHMEM communication contexts extension provides a mechanism for isolating streams of communication operations; the context is provided as an argument to every SHMEM communication and synchronization operation. Thus, operations from different threads within a PE can be isolated by using separate contexts for each thread. Likewise, both single-threaded and multithreaded PEs can perform sets of nonblocking operations using different contexts, allowing them to be completed as separate batches, providing better overlap.

Communication contexts are transmit-side constructs that are applied to point-to-point communication operations. Contexts do not alter the existing structure of OpenSHMEM symmetric memory, symmetric object creation, or collective operations. Contexts behave similarly to request objects that are used by many communication libraries to manage completion of individual operations. An important difference is that contexts are intended to also provide a mapping to per-context communication resources in multithreaded executions, thus providing thread isolation when each thread uses its own context.

3.1 Application Programming Interface

The OpenSHMEM API extension to support contexts is shown in Listing 1. Context creation and destruction operations are local to the calling PE; that is, these operations are not collective. Multiple contexts can be created at once for convenience. An optional assertion argument can be used to assert a restricted usage model, potentially enabling the implementation to operate on the given context more efficiently. For example, if a context will be used by a single thread in a multithreaded PE, the user can supply the value SHMEM_THREAD_SERIALIZED as the assertion argument, enabling the runtime system to eliminate internal synchronizations.

3.1.1 Communication Operations

New bindings are added for all one-sided get, put, and atomic operations. We show a subset of these operations in Listing 1 for conciseness. In particular, we show the nonblocking variants of get, put, and integer fetch-and-add (FADD) operations since they represent a broad set of usage models. However, blocking variants of communication operations are also added. All operations require an additional context argument that indicates the context in which the operation should be performed. A special context, called SHMEM_CTX_DEFAULT is added and can be supplied as the context argument in any operation. Existing operations that do not have a context argument are defined to operate on this context.

3.1.2 Completion and Ordering of Operations

Context quiet and fence operations are added to provide remote completion and point-to-point ordering, respectively, at the level of individual contexts. A context quiet or fence must provide the desired completion or ordering semantic only for the context on which the operation was performed. A call to the existing shmem_quiet() or shmem_fence() operation impacts only on the

```
/* Creation and destruction routines (local) */
int  shmem_ctx_create(int num_ctx, int assertion, shmem_ctx_t ctx[]);
void shmem_ctx_destroy(int num_ctx, shmem_ctx_t ctx[]);

/* Communication routines (one-sided, list abbreviated) */
void shmem_ctx_putmem_nb(shmem_ctx_t ctx, void *target, const void *source, size_t len, int pe);
void shmem_ctx_getmem_nb(shmem_ctx_t ctx, void *target, const void *source, size_t len, int pe);
int  shmem_ctx_int_fadd_nb(shmem_ctx_t ctx, int *target, int value, int pe);
...

/* Completion and ordering routines (one-sided) */
void shmem_ctx_fence(shmem_ctx_t ctx);
void shmem_ctx_quiet(shmem_ctx_t ctx);

/* Synchronization routines (collective) */
void shmem_sync(int PE_start, int logPE_stride, int PE_size, long *pSync);
void shmem_sync_all(void);
```

Listing 1: OpenSHMEM communication contexts Application Programming Interface (API), C programming language.

SHMEM_CTX_DEFAULT context. If a second thread performs communication operations on a context in parallel with a quiet or fence operation on that same context, the new communication operations are not guaranteed to be completed or ordered by the quiet or fence operation.

3.1.3 PE Synchronization

The existing shmem_barrier() and shmem_barrier_all()[1] operations include an implicit quiet operation, which is performed on the default context. We also add a new barrier construct, which we call shmem_sync(). This so-called "loud" barrier does not include an implicit quiet; it requires the user to synchronize contexts explicitly. This enables the user to synchronize all PEs while allowing nonblocking communication issued before the barrier to continue in the background beyond the barrier, e.g. to pipeline several stages or iterations of a parallel computation. Like the barrier operation, sync does not offer any inter-PE synchronization between threads.

3.2 Example Code

We show a simple hybrid SHMEM+OpenMP program in Listing 2. In this program, the OpenSHMEM library is initialized in the SHMEM_THREAD_MULTIPLE mode, enabling multiple threads in a PE to perform SHMEM operations concurrently. After initializing the library, the PE creates contexts for each thread that will exist within the OpenMP parallel region. Contexts are created with the SHMEM_THREAD_SERIALIZED assertion to inform the OpenSHMEM runtime system that each will be used by a single thread.

Within the OpenMP parallel region, threads query their thread identity and cache it in the tid variable. This thread-local state is stored outside of the OpenSHMEM library in a variable that is private to each thread. The thread ID is used to identify the thread's context in the context array. Within the parallel region, threads perform computation followed by a communication step. After issuing multiple nonblocking put operations, each thread performs a context quiet operation. The context quiet operation completes only the operations issued by that thread, allowing threads to proceed independently.

4. IMPLEMENTATION OF CONTEXTS

Contexts provide a way to complete a subset of pending communication. This requires that the OpenSHMEM implementation have a way to track outstanding and completed communication operations independently for each context. To accomplish that, the implementation needs some form of aggregate handles for each context. In one extreme, the implementation might keep a per-operation handle, but that is likely to incur too much overhead. Here we discuss a particular implementation that uses Portals counters as a form of aggregate handles.

4.1 Implementation on Portals 4

A block diagram of our context implementation over the Portals 4 interface is shown in Figure 1. The per-context resources required are a Portals counter (CT) object, a Portals memory descriptor and a local counter (i.e. an 8-byte variable) held by the runtime system. Portals uses memory descriptors (MDs) to drive all of the communication, and one can attach a Portals counter to the memory descriptor to count the number of remote completions (ACKs) received through counting events.

The Portals OpenSHMEM runtime system uses the local counter to track all of the communication operations issued on each context. The local counter must be atomically incremented before the call to the Portals library that issues the communication. Because of the Portals end-to-end reliability model, when the SHMEM library needs to guarantee remote completion of any or all of the communication calls, it must wait until the Portals counter (i.e., the ACK counter) reaches the value of the local counter. This blocking can occur during a fence or quiet operation, when the SHMEM library blocks until all of the communication has completed remotely. It can also occur on a blocking communication call (e.g. large put or get) when the library blocks until all communication completes remotely. Waiting for all operations to complete remotely is necessary in this case, because it is not possible to determine which operation has completed when Portals counter is incremented. Ideally, for blocking communication one would like to only complete the current communication call, instead of waiting for all of the remote completions. We discuss the reasoning for this implementation decision in Section 4.2.

OpenSHMEM blocking communication calls, require only a local completion indicating that buffers can be read or reused. Small puts are commonly buffered by the networking layer or the runtime system, allowing them to return immediately. However, for large puts (e.g. larger than the Portals volatile size) the local and remote completions become equivalent in the presence of message delivery guarantees. The sender can safely release its buffer only after it

[1]The "all" variant of barrier incorporates participation from all PEs, whereas shmem_barrier() function can operate on a subset of PEs.

```
int main(int argc, char **argv) {
    int max_threads = omp_get_max_threads();
    shmem_ctx_t thread_ctx[max_threads];

    shmem_init(SHMEM_THREAD_MULTIPLE);
    shmem_ctx_create(max_threads, SHMEM_THREAD_SERIALIZED, thread_ctx);

#pragma omp parallel
    {
        int tid = omp_get_thread_num();

        /* Threads operate on their private (SERIALIZED) contexts */
        while (!done) {
            /* Perform computation */
            ...

            /* Perform communication */
            for (i = 1; i < npes; i++)
                shmem_ctx_putmem_nb(thread_ctx[tid], ..., (my_pe + i) % npes);

            shmem_ctx_quiet(thread_ctx[tid]);
        }
    }
    shmem_ctx_destroy(max_threads, thread_ctx);
    shmem_finalize();
}
```

Listing 2: Hybrid OpenSHMEM+OpenMP communication contexts example.

knows said buffer is not needed for retransmission, which means the sender must wait for the remote completion before issuing the local completion.

All non-context SHMEM operations are internally performed as context operations on SHMEM_CTX_DEFAULT context. Additionally, to simplify the implementation, all SHMEM operations are internally implemented as non-blocking. When a blocking SHMEM call is made, internally the implementation translates it to a non-blocking call followed by an appropriate completion operation. In many cases the non-blocking completion is a call to shmem_quiet(), but sometimes — e.g. in case of small messages whose size is below the Portals volatile limit, indicating that the data has been buffered by the Portals layer — that is not necessary. As a result, these implementation decisions yield very efficient non-blocking context operations.

4.2 Minimizing Sender-Side Resources

Our context implementation uses a minimal set of resources, while still providing the described context functionality. The minimum required resource for a Portals 4 implementation are: one Portals counter that keeps track of how many operations have completed remotely; one Portals memory descriptor (MD) that is used to identify the sender side Portals counter; and one local counter (i.e. an 8-byte variable) that counts the number of issued communication operations. The benefits of using fewer resources are that the application is free to use more contexts without exhausting the available resources. However, some of the downsides have to be considered when writing applications. For example, completing get operations (as well as fetch atomics) also requires remote completion of all blocking and non-blocking put operations made on the same context. Because there is only one available Portals counter per context, it is not possible to distinguish get from put operations, and when one of them must be remotely completed (e.g. using a quiet operation), the implementation must wait for all of the pending operations to complete. There is a similar potential performance pitfall when using blocking and non-blocking operations. A blocking operation that follows non-blocking operations on the same context might also force the completion of all of the pending non-blocking operations. If these performance concerns arise, we suggest creating separate contexts for blocking and non-blocking operations.

A possible alternative implementation that does not suffer from the described performance caveats would internally use multiple counters per context to separate the gets from the puts and blocking from non-blocking operations. Alternatively, full events rather than counting events could be generated for fetching operations. While convenient, such alternative implementation use more of the scarce counter resources than necessary. Therefore, we choose the leaner approach that uses fewer resources, and leave it up to the application programmer to address the specific performance concerns if they arise. A very simple way to deal with the described performance issues is to simply create an additional context that is used for fetching operations.

5. PERFORMANCE EVALUATION

Contexts are evaluated on two benchmarks: Mandelbrot and Integer Sort (IS). Mandelbrot is a multithreaded benchmark that computes the complex-plane points that are members of the Mandelbrot set [20]. Integer sort is an OpenSHMEM version of IS benchmark from NAS Parallel Benchmark suite [1, 10] that implements parallel bucket sort. Both benchmarks run over the Portals-SHMEM implementation of OpenSHMEM [13], which has been extended to include contexts. The Portals-SHMEM implementation itself was run using the Portals 4 over InfiniBand* reference implementation [16]. Our analysis is focused on relative performance gains from using contexts because the absolute numbers heavily depend on the available Portals 4 implementation.

Experiments were conducted on a 16-node cluster with a Mellanox* QDR InfiniBand interconnect. Each node in this cluster is configured with 24GB of memory and two Intel® Xeon® X5680 processors, for a total of 12 cores per node, each supporting two hyperthreads, for a total of 24 hardware threads per node. Because the

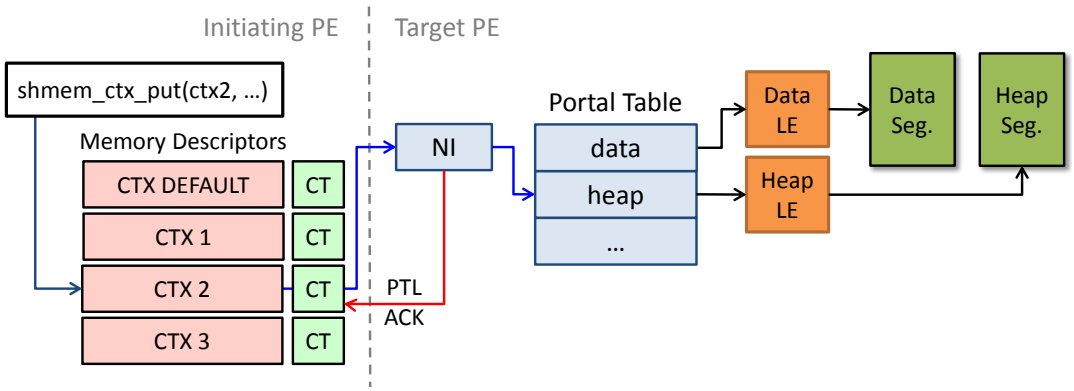

Figure 1: Portals 4 implementation of communication contexts, showing Portals objects. Contexts are backed by a counting event (CT) at the sender, which is associated with a memory descriptor (MD). Upon completion of a one-sided operation, the sender's counter is incremented by an acknowledgement generated by receiver's Network Interface (NI).

```
while (!work_completed()) {
  int dest_pe = next_round_robin_pe();
  long my_job = shmem_long_fadd(
         &job_counter, JOB_SIZE, dest_pe);

  /* Check if work at dest_pe is done */
  if (my_job > JOB_SIZE*JOBS_PER_PE){
    mark_done(dest_pe);
    continue;
  }

  for (i = 0; i < JOB_SIZE; i++)
    buf[i] = compute_point(my_job + i);

  shmem_putmem(&mandel_data[my_job],
               buf, JOB_SIZE, dest_pe);
}
```

Listing 3: Mandelbrot set benchmark pseudo-code.

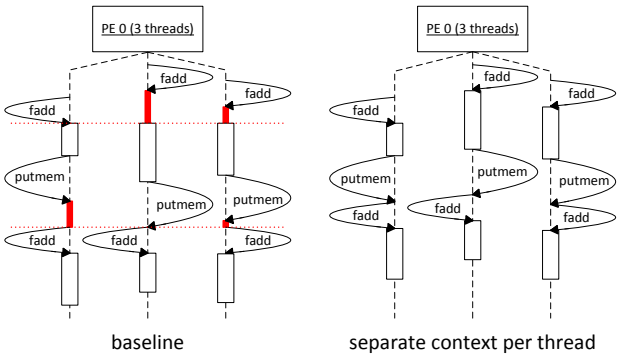

Figure 2: Mandelbrot timing diagram comparison with no-contexts (left) and with one context per thread (right).

Portals-IB implementation generates two communication threads, we limit our multithreaded experiments to 11 cores and pin these threads to the 12^{th} core. For single-threaded experiments, we limit our experiments to 6 PEs per node to avoid measuring oversubscription overheads.

5.1 Mandelbrot Set Benchmark

Simplified pseudo-code for the Mandelbrot set benchmark is shown in Listing 3. This benchmark is capable of running in multi-threaded mode, and the expected usage is to run one process per node with one thread per processor core. Each thread independently runs the algorithm shown in Listing 3.

Domain decomposition is used to distribute the grid of calculated points evenly across PEs. On top of that decomposition, a simple distributed load-balancing algorithm is implemented. Load-balancing is necessary because the Mandelbrot set point calculation times vary significantly between points. The implementation of the distributed load-balancing algorithm is shown in Listing 3. The outer loop communicates with all PEs in a round-robin manner, to avoid hotspots. As a part of the load balancing algorithm, the benchmark first fetches a chunk of work by issuing atomic fetch-and-add to a remote counter. For each point in the fetched chunk of work, the local thread computes whether that point is in the Mandelbrot set. After the computation is complete, the results are sent to the PE that issued the work.

Multiple threads that run at the same time interfere with each other. Consider what happens when all threads make SHMEM calls using the same global resources (i.e. the no-contexts case). Because there are no individual contexts (i.e. separate Portals counters) for each thread, all ongoing communication in all threads must be completed before each thread can be certain its own communication has completed. This has potential to cause serious performance issues which become more severe with an increasing number of threads. An example using three threads interfering with each other is shown on the left in Figure 2. Assigning a separate context to each thread solves this problem because it enables each thread to independently complete communication operations, as shown on the right of Figure 2.

Using one context per thread eliminates inter-thread interference in the SHMEM library; contexts can also provide additional opportunities to improve performance within each thread by overlapping communication with computation. The default case that uses one context per thread is shown on the left of Figure 3. In this example, one of the threads in PE 0 requests work from PE 1 using fetch-and-add, does the computation, and then returns the results back to PE 1. After the results are sent back to PE 1, it is ready to request more work from PE 2 and the process repeats. There is an oppor-

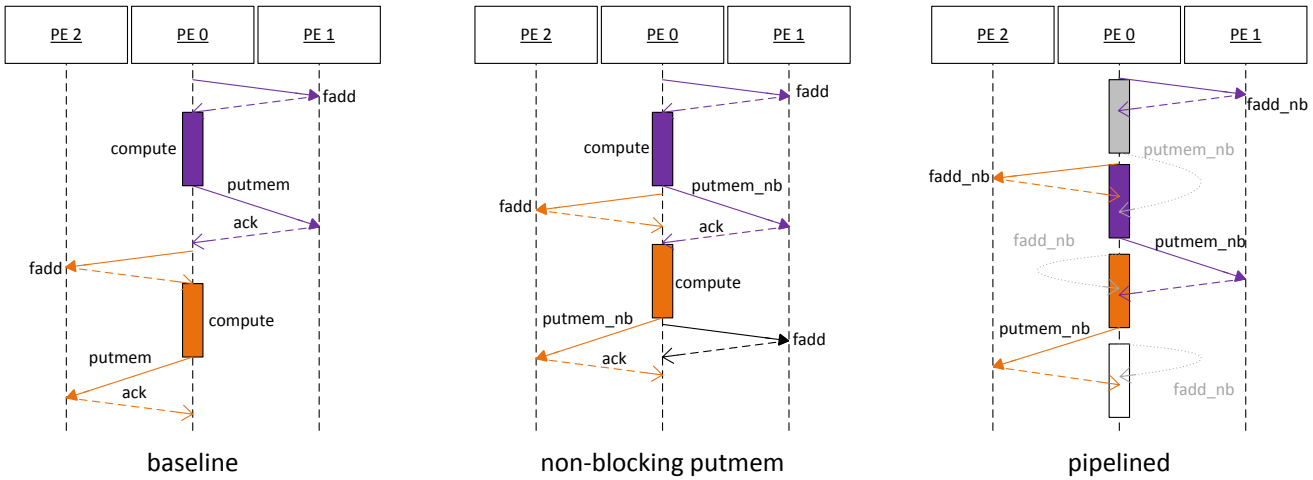

Figure 3: Mandelbrot timing diagram comparison of computation/communication overlap scenarios.

tunity here to overlap the putmem and fetch-and-add by using the nonblocking version of the putmem. The nonblocking approach shown in the middle of Figure 3 improves performance, but there is yet more opportunity to overlap the communication with the computation phase.

A pipelined version of the Mandelbrot set benchmark shown on the right of Figure 3 achieves the best performance by overlapping communication and computation. To enable this overlap, each thread uses two separate contexts. In the first pipeline stage the benchmark issues a nonblocking fetch-and-add to the next round PE (as opposed to the current PE). This fetch-and-add is completed later in the next round by the second stage, but by that time the expectation is that the response to the fetch-and-add is already available. This response is then used to identify the chunk of work that needs to be computed. At the end of the second stage, the PE sends the computed results using a nonblocking putmem.

The pipelined implementation requires two contexts per thread and two copies of the local compute buffer. The implementation switches between the two contexts and the compute buffers at the end of each iteration. Nonblocking fetch-and-add is completed using shmem_ctx_quiet(), which also completes the nonblocking putmem, thus freeing one of the local compute buffers to be reused.

Performance gains provided by each of the described optimizations are shown in Figure 4. The x-axis shows the number of processor cores used on each node, which corresponds to the number of threads in the threaded cases. In the non-threaded case that only uses processes, the x-axis corresponds to the number of processes on a single node. The y-axis shows the rate of calculating which points belong to the Mandelbrot set. All experiments are run using 16 cluster nodes.

Adding one context per thread without any other code changes improves performance by 13% for the case of running 11 threads per node. If the nonblocking putmem interface is also used, the performance improves by an additional 5%. Contexts also enable computation and communication overlap by using pipelining. Pipelining requires two contexts per thread and two local compute buffers, but it provides over 25% performance improvement over the baseline no-context case for 11 threads.

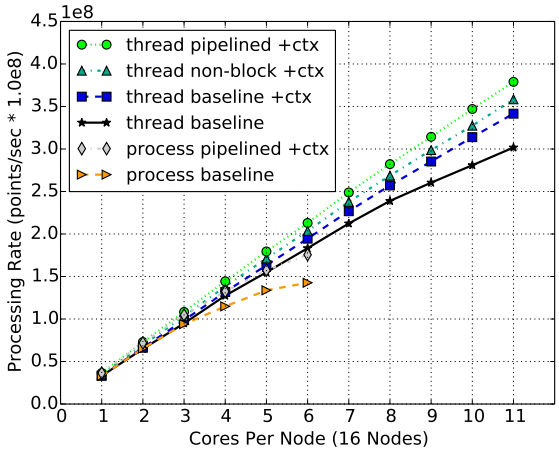

Figure 4: Strong scaling performance of Mandelbrot benchmark.

Contexts also provide improved scaling efficiency for the Mandelbrot benchmark. Figure 5 shows that implementations using contexts scale much better than the baseline case without contexts, because contexts provide thread separation inside the SHMEM library, and thus reduce interference between threads. There is potential for even better scaling if the separation that contexts provide is extended to layers underneath SHMEM; in our case to the Portals 4 over InfiniBand implementation.

Figures 4 and 5 also compare non-threaded implementations that rely solely on processes. We were only able to scale up to 6 processes per node because each SHMEM PE also requires its own Portals progress thread, and sharing core resources between threads would yield noisy results. This is also the reason why we only run up to 11 threads per node, with the 12th core being dedicated to the Portals progress thread. Surprisingly, the non-threaded implementation showed much poorer scaling than the threaded implementation. We are observing much higher than expected number of instruction cache misses as we scale the non-threaded benchmarks, but the exact cause for that is still under investigation.

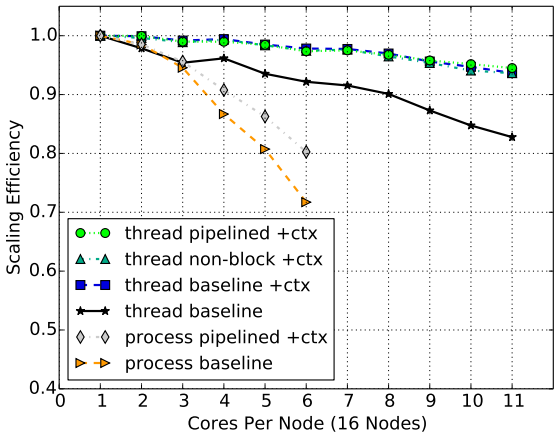

Figure 5: Scaling efficiency of Mandelbrot benchmark.

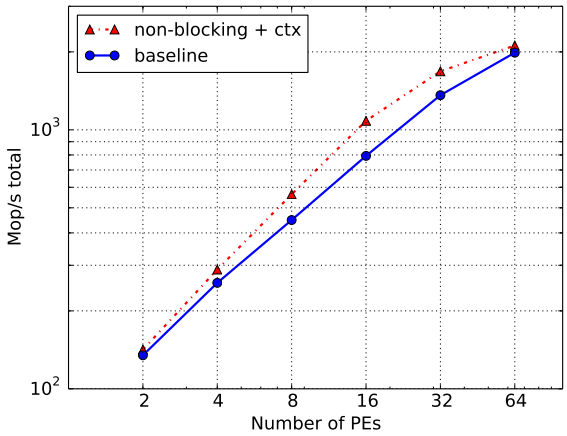

Figure 6: Performance of integer sort benchmark.

```
shmem_barrier_all();

for (i = 0; i < num_pes; i++) {
  int k1 = send_offset[i];
  int k2; // target offset

  shmem_ctx_int_get(ctx[0], &k2,
            &recv_offset[me], 1, i);

  shmem_ctx_int_put_nb(ctx[1],
            key_buff2+k2, key_buff1+k1,
            send_count[i], i);
}
shmem_ctx_quiet(ctx[1]);
shmem_barrier_all();
```

Listing 4: Integer sort benchmark, key exchange loop snippet.

5.2 Integer Sort Benchmark

The integer sort benchmark is a parallel bucket sort code from the NAS Parallel Benchmark suite [1]. All runs were done using the "C" workload class, which sorts 134 million integer keys over all PEs. The benchmark has an all-to-all communication pattern where each PE sends a bucket of keys to every other PE. However, before sending the keys to each target PE, the sender must obtain the target location from the target PE. The code snippet for this key exchange loop is shown in Listing 4. The implementation in the listing uses two contexts, one for the get and one for the nonblocking put. This allows the nonblocking puts to happen in the background, as they are not completed until the shmem_ctx_quiet() call after the loop.

Performance benefits from using contexts in this benchmark are shown in Figure 6. The performance gains are expected to be higher as the number of PEs increases due to the all-to-all nature of communication. We ran the experiments on a 16 node cluster, with multiple PEs per node for the 32 and 64 PE data points. Beyond 16 PEs our gains start to show diminishing returns because the benchmark performance becomes limited by PEs sharing the underlying InfiniBand network adapter, and because of the suboptimal incast communication schedule used in the IS benchmark's implementation. However, we still observe a good performance increase of 35% for the 16 PE case.

6. DISCUSSION

As with all middlewares, OpenSHMEM must provide alignment between the application and the underlying system, while meeting portability and performance requirements. We next discuss several tradeoffs and design decisions related to communication contexts and analyze the impact of this new interface on both applications and the underlying networking layer.

6.1 Design Choices

The default communication context. In Section 3 we introduced the SHMEM_CTX_DEFAULT context, which is the default context in which non-context operations (e.g. a call to the existing shmem_putmem() routine) are performed. Non-context quiet, fence, and barrier operations were defined to impact only operations performed in the default context.

An alternative semantic for non-context quiet, fence, and barrier operations is that operations performed on all contexts are synchronized. This behavior could be useful to application developers by providing a convenient mechanism to complete or order all outstanding communication operations. However, such a semantic defeats the isolation property that we wish to provide for multi-threaded SHMEM executions. Supporting operations that impact all contexts would require the OpenSHMEM runtime system to maintain an internal list of all contexts and perform the operation on all contexts in this list. Accesses to the list would need to be synchronized, as the list would be modified by context creation and destruction operations. While efficient synchronization techniques exist, the desire to eliminate such overheads motivated the choice to isolate all non-context operations to the default context.

Dependence on threading package. The "+X" in hybrid OpenSHMEM+X parallel programming provides an important conduit for innovation in node-level parallel programming models, runtime systems, and architectures. Thus, we have endeavored to create an interface that enables an OpenSHMEM runtime system implementation that is independent of the particular node-level parallel programming model in use. This is achieved by the addition of an explicit shmem_ctx_t context object to the API. The context object is provided by the calling thread in every point-to-point operation and all context-specific information is stored within the context

object, eliminating dependencies on Thread-Local Storage (TLS).

6.2 Implications to Network Stack

In a single-threaded SHMEM execution, every PE is provided with a separate set of communication resources used to interface with the network. These resources typically include transmit command queues, event queues, and structures for processing incoming messages. In contrast to single-threaded executions, all threads in a multithreaded SHMEM execution typically share the same set of resources. This sharing can incur synchronization overheads and encounter resource exhaustion overheads.

SHMEM communication contexts are intended to provide a mechanism that enables threads to bring the same set of resources to bear on communication as PEs; thus enabling multithreaded executions to achieve the same level of communication performance as single-threaded executions. In order to support this, the runtime system must be able to back each context with an independent resource set. While some networking layers fully support this resource mapping, challenges still exist for many existing networking layers.

6.2.1 Resource Management in Portals 4

In the Portals 4 network programming interface, the Network Interface (NI) object provides the resource abstraction that is visible through the API. One option for supporting contexts is to open multiple NIs, one per thread. This would require the Portals implementation to define a special set of Portals interface names that can be used to open the same interface multiple times by threads in the same process. A benefit of this approach is that additional resources – including Portals counters – would become available to every thread. However, NIs are also addressable for the purposes of communication and opening more NIs than SHMEM PEs in the job introduces an addressing problem. By default, NIs are physically addressable using their node ID (NID) and process ID (PID). The addressing problem can potentially be resolved by assigning specific PIDs per node to act as receivers for messages destined to a particular SHMEM PE on that node. Given such a mapping, a Portals logical address mapping can be created that maps integer PE ids to the designated NID and PID.

A Portals Memory Descriptor (MD) also represents a sender-side resource that is provided in every communication operation. The MD manages the sender-side buffers used in communication and is also used to track the completion of individual communication operations. Thus, the MD provides a good mechanism for individually tracking the stream of operations corresponding to a given context. However, Portals currently does not provide a mechanism for indicating that an MD should interface with a separate set of message injection resources in the underlying networking layer.

In order to indicate that MDs corresponding to communication contexts should be mapped to individual message injection ports in the underlying network, a new MD attribute PTL_MD_INDEPENDENT could be added. This attribute indicates to the Portals implementation that messages sent from the given MD represent a stream of operations that is independent of operations performed on other MDs. When possible, the Portals implementation should provide this memory descriptor with a separate set of resources for issuing communication operations.

The amount of available Portals resources, such as counters and MDs, depends on the implementation and is returned to the user during Portals library initialization. In our reference Portals-SHMEM implementation there are 1024 counters and 1024 MDs available to each PE. This sets the theoretical number of contexts per PE to 1024, however if these counters and MDs are also used for other functionality, that would reduce the maximum possible number of contexts. Any of the contexts can be used by any thread, and if the context has been created with the SHMEM_THREAD_MULTIPLE hint, multiple threads can share that context.

7. RELATED WORK

A SHMEM thread safety extension was proposed by ten Bruggencate et al. [19]. This proposal captures existing thread safety extensions that are provided by the Cray* SHMEM library. In addition, it proposes several new extensions that integrate threading with the SHMEM library, enabling each thread to generate an independent stream of communication operations.

Thread safety adds the new shmem_init() function, which allows the user to select the level of threading support that will be used for the job (e.g. SHMEM_THREAD_MULTIPLE). In addition, it defines the conventions and practices that must be followed in order to achieve correct behavior when multithreaded PEs are used. Using contexts with multithreaded PEs requires thread safety, which can be provided through this set of extensions.

Thread integration adds new functions to register threads with the SHMEM library and synchronize the operations performed by individual threads. In addition, it provides several functions, such as a thread barrier, that can be used to coordinate among the registered threads in a PE. This proposal requires a much smaller change to the OpenSHMEM API than contexts because it does not introduce new versions of the communication routines. Because a context is not specified in each operation, the proposed interface does not provide a means for threads to generate independent streams of communication operations. In addition, it requires that the SHMEM runtime system identify the calling thread in each communication operation in order to associate the operation with a specific thread. Such Thread-Local Storage (TLS) lookups are often expensive relative to the latency of a communication operation, and also require that the SHMEM runtime system to be able to utilize the TLS system provided by the threading package that will be used. In contrast, contexts are designed to be independent of the threading package; a reference to the context object is passed by the user in every operation and any context-specific information can be stored within this object.

The Message Passing Interface (MPI) [9] utilizes communicators and windows as communication contexts for two-sided and one-sided communication, respectively. The MPI endpoints extension [6, 18] is intended to address similar challenges in the context of MPI by providing additional ranks that can be assigned to threads. A significant difference between endpoints and contexts is that endpoints are individually addressable in MPI communication operations. In contrast, contexts are a sender-side only construct in OpenSHMEM and communication operations still target PEs, rather than individual contexts or threads within a PE.

When processes are used on every core, most networks provide each process with a separate set of communication resources (e.g. command queues); however, when threads are used, all threads typically share the set of resources provided to the parent process. Both OpenSHMEM contexts and MPI endpoints strive to address this shortfall for hybrid programming by providing a high-level construct that can be mapped to a lower-level resource set. Some networking APIs provide an abstraction for this resource set, such

as UCCS Contexts [17] and IBM PAMI* endpoints [8].

Contexts also provide a means for aggregate tracking of outstanding communication operations. This aspect of contexts is similar to aggregate nonblocking communication handles in ARMCI [11] and GasNet [4]; communication queues in GASPI [7]; and communication contexts in PAMI Contexts [8].

8. CONCLUSION

The OpenSHMEM communication contexts extension provides a mechanism for isolating individual streams of communication operations. Contexts aim to improve communication-computation overlap in conventional SHMEM executions and also to provide isolation for threads in multithreaded SHMEM executions. In addition, contexts provide a mechanism that can be used by the OpenSHMEM runtime system to map threads to individual sets of network resources. It is believed that such mechanisms are key components in the design of runtime systems that can efficiently support multithreaded communication.

We presented an implementation of communication contexts using the Portals 4 network programming interface and evaluated its performance using a hybrid SHMEM+Threads Mandelbrot set benchmark and a single-threaded NAS integer sort benchmark. Through the addition of contexts, we observed a 25% performance improvement for Mandelbrot by eliminating thread interference and enabling pipelining, and a 35% performance improvement for IS by enabling more effective overlap of communication and computation.

9. ACKNOWLEDGMENTS

We thank the members of the OpenSHMEM and Portals communities for many insightful discussions that helped us to refine the OpenSHMEM communication contexts interface and its mapping to the Portals network programming interface.

*Other names and brands may be claimed as the property of others.

10. REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[2] R. Bariuso and A. Knies. Shmem user's guide. Technical Report SN-2516, Cray Research, Inc., 1994.

[3] B. W. Barrett, R. Brightwell, K. S. Hemmert, K. T. Pedretti, K. B. Wheeler, and K. D. Underwood. Enhanced support for OpenSHMEM communication in Portals. In *Hot Interconnects*, pages 61–69. IEEE, 2011.

[4] D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.

[5] Cray Research, Inc. *SHMEM Technical Note for C*, 1994. SG-2516 2.3.

[6] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling communication concurrency through flexible MPI endpoints. *To Appear in Intl. J. High Performance Computing Applications (IJHPCA)*, Dec. 2013.

[7] GASPI Consortium. GASPI: Global address space programming interface specification of a PGAS API for communication. Version 1.00, June 2013.

[8] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A parallel active message interface for the Blue Gene/Q supercomputer. In *Proc. 26th Intl. Parallel Distributed Processing Symposium*, IPDPS '12, pages 763–773, May 2012.

[9] MPI Forum. MPI: A message-passing interface standard version 3.0. Technical report, University of Tennessee, Knoxville, Sept. 2012.

[10] NAS parallel benchmarks for OpenSHMEM, version 1.0a. Online: `http://bongo.cs.uh.edu/site/Downloads/Examples`, Aug. 2014.

[11] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586, 1999.

[12] *OpenSHMEM Application Programming Interface, Version 1.1*, June 2014.

[13] OpenSHMEM implementation using portals 4. Online: `http://code.google.com/p/portals-shmem/`, Aug. 2014.

[14] S. W. Poole, O. Hernandez, and P. Shamis, editors. *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - First Workshop, OpenSHMEM 2014, Annapolis, MD, USA, March 4-6, 2014. Proceedings*, volume 8356 of *Lecture Notes in Computer Science*. Springer, 2014.

[15] S. W. Poole, P. Shamis, A. Welch, S. Pophale, M. G. Venkata, O. Hernandez, G. A. Koenig, T. Curtis, and C.-H. Hsu. OpenSHMEM extensions and a vision for its future direction. In Poole et al. [14], pages 149–162.

[16] Portals 4 open source implementation for InfiniBand. Online: `http://code.google.com/p/portals4/`, Aug. 2014.

[17] P. Shamis, M. Venkata, J. Kuehn, S. Poole, and R. Graham. Universal common communication substrate (UCCS) specification, version 0.1. Technical Report ORNL/TM-2012/339, Oak Ridge National Laboratory, 2012.

[18] S. Sridharan, J. Dinan, and D. Kalamkar. Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints. In *Proc. 26th Intl. Conf. for High Performance Computing, Networking, Storage, and Analysis*, SC, Nov. 2014.

[19] M. ten Bruggencate, D. Roweth, and S. Oyanagi. Thread-safe SHMEM extensions. In Poole et al. [14], pages 178–185.

[20] Wikipedia. Mandelbrot set — wikipedia, the free encyclopedia. Online: `http://en.wikipedia.org/w/index.php?title=Mandelbrot_set`, Aug. 2014.