Simplifying Communication Overlap in OpenSHMEM Through Integrated User-Level Thread Scheduling

Md. Wasi-ur- Rahman\(^1\(\text{coauthor}\)), David Ozog\(^2\), and James Dinan\(^2\)

\(^1\) Intel Corporation, Austin, TX, USA
md.rahman@intel.com
\(^2\) Intel Corporation, Hudson, MA, USA

Abstract. Overlap of communication with computation is a key optimization for high performance computing (HPC) applications. In this paper, we explore the usage of user-level threading to enable productive and efficient communication overlap and pipelining. We extend OpenSHMEM with integrated user-level thread scheduling, enabling applications to leverage fine-grain threading as an alternative to non-blocking communication. Our solution introduces communication-aware thread scheduling that utilizes the communication state of threads to minimize context switching overheads. We identify several patterns common to multi-threaded OpenSHMEM applications, leverage user-level threads to increase overlap of communication and computation, and explore the impact of different thread scheduling policies. Results indicate that user-level threading can enable blocking communication to meet the performance of highly-optimized, non-blocking, single-threaded codes with significantly lower application-level complexity. In one case, we observe a 28.7% performance improvement for the Smith-Waterman DNA sequence alignment benchmark.

1 Introduction

Communication latency hiding through pipelining and overlap with computation are key optimizations for High Performance Computing (HPC) applications. Popular communication middleware, such as MPI [22] and OpenSHMEM [24], facilitate these optimizations through non-blocking communication; however, managing asynchronous data movement can lead to significant application-level complexity. Multi-threaded programming can provide a simpler approach to enabling communication asynchrony, but the over-subscription required to reach effective pipelining depths can result in high context switching overheads in typical multi-threaded environments.

User-level threading and tasking models have been proposed as alternatives to conventional operating system (OS) scheduled threads. In contrast with OS threads, user-level threads reduce context switching overheads through cooperative, rather than preemptive, thread scheduling that is performed at the user level, without invoking the OS thread scheduler. Thus, relative to OS level multi-threading approaches, user-level threading enables a greater number of fine-grain operations to be in-flight. By reducing...
these overheads, user-level threading can make feasible multithreaded approaches to hiding communication latency.

Version 1.4 of the OpenSHMEM specification [24] was recently ratified and introduced threading support that allows OpenSHMEM programs with multi-threaded processes (PEs). This new feature enables hybrid programming with OpenSHMEM, which can be exploited to enable on-node shared memory programming and to enable trade-offs between the number of PEs and number of threads on a given node. However, in traditional usage, hybrid programming has avoided over-subscription, because of the overheads associated with context switching.

Figure 1(a) illustrates the performance challenges associated with multi-threading using a bandwidth experiment where 8 sender and receiver PEs are involved in measuring the uni-directional streaming bandwidth through the $\text{shmem\ put}$ operation. Further details of the experimental setup are available in Sect. 5. In this experiment, we use both blocking and non-blocking versions of the API and also launch the blocking API test with multiple OpenMP [6] threads. As shown in Fig. 1(a), while 4 OpenMP threads with blocking APIs improve the bandwidth achieved for several message sizes compared to the non-blocking API experiment, 16 OpenMP threads cause the performance to degrade relative to the blocking API experiment because of the over-subscription overheads.

A challenge to user-level threading is that such models require explicit, cooperative scheduling of threads and deadlock can occur if threads block in OpenSHMEM operations without first yielding. However, user-level control results in significantly lower overheads, as shown in Fig. 1(b), which compares user-level and OS thread creation and context switch operation costs. In this comparison, user-level threads are created using the Unix* $\text{ucontext}$ [1] interface. The context switch overhead is measured by averaging a total of 100,000 context switches between two participating threads using a synthetic benchmark [8]. This experiment highlights that user-level threading can significantly reduce overheads.

In this work, we leverage these insights to design a generic thread scheduling extension to OpenSHMEM that integrates user-level threading with the OpenSHMEM middleware, enabling the runtime system to perform cooperative thread scheduling when threads become blocked in communication operations. One of the major challenges in
our implementation of this model is to detect the appropriate threads for scheduling that will ensure effective forward progress of the application. In this regard, we propose communication-aware thread scheduling in our extension, which reduces overheads by avoiding threads that remain blocked on pending communication. We extend the open source Sandia OpenSHMEM [30] library to support user-level thread scheduling and evaluate our approach using several applications in conjunction with the popular Argobots [3] user-level threading system. Results indicate that, while threading overheads are still present, user-level threading can enable communication overlap comparable to that achieved with non-blocking communication and at a much lower level of complexity in application code. For the Smith-Waterman DNA sequence alignment benchmark, we observe 28.7% performance resulting from the addition of user-level threading.

2 Background and Related Work

We begin with a summary of the OpenSHMEM library specification, focusing on the recent developments that define multi-threaded interfaces, which are most relevant to this paper. For additional detail on the OpenSHMEM APIs, we refer the reader to the OpenSHMEM specification [24]. Next, we review user-level threading and prior approaches to thread integration.

2.1 OpenSHMEM

OpenSHMEM [24] is a community specification that defines a Partitioned Global Address Space (PGAS) parallel programming model. An OpenSHMEM application is comprised of multiple processes (PEs) running the same program, where each PE is parameterized with a unique integer identity in the range 0\ldots \text{npes} – 1. PEs expose a symmetric data segment and a symmetric heap for remote access. The values in memory at each PE differ; however, the layout of the symmetric segments is identical at all PEs, simplifying usage and providing opportunities for implementations to optimize performance. OpenSHMEM defines a library API that enables asynchronous, one-sided access to symmetric data at all PEs through put/get data transfers, atomic operations, and collective communication primitives.

The recently released OpenSHMEM 1.4 specification introduced support for multi-threaded communication with OpenSHMEM routines. Similar to multi-threading in MPI, OpenSHMEM provides an initialization routine, \text{shmem\_init\_thread}, which allows the user to specify a level of thread support required by the application. The most restrictive thread level is \text{SHMEM\_THREAD\_SINGLE}, for which there must \textit{not} be any threads used by the application; and the least restrictive thread level supported is \text{SHMEM\_THREAD\_MULTIPLE} for which \textit{any} thread may call an OpenSHMEM routine at any time. There is also a \text{SHMEM\_THREAD\_FUNNELED} model in which only a main thread invokes OpenSHMEM routines and a \text{SHMEM\_THREAD\_SERIALIZED} model, in which the application serializes OpenSHMEM calls made by any application thread.

OpenSHMEM 1.4 also introduced a communication \text{contexts} API, which facilitates better overlap of communication and computation by enabling applications to express streams of operations that can be synchronized and ordered independently. The contexts
API enables implementations to isolate groups of threads, reduce internal threading overheads, and more effectively manage underlying communication resources [14].

2.2 Sandia OpenSHMEM and OFI

We define a generic extension to OpenSHMEM to support an arbitrary user-level thread or task system focusing on the open source Sandia OpenSHMEM (SOS) implementation [30] that uses the OpenFabrics Interfaces (OFI) libfabric communication library [16] to support multiple popular HPC networks. Detailed descriptions of the implementation of SOS on OFI can be found in [31].

The OFI libfabric communication library provides a common, low-level interface to high-speed networks. OFI’s design is focused on portable support for HPC communication, which requires low latency and high throughput. OFI defines a complete set of interfaces that enable one-sided and two-sided messaging, memory registration, communication event management, collective communication, and many other features. A primary focus of this work are OFI communication event counters that are used in SOS to track the number of communication operations pending on a given context.

2.3 User-Level Thread Libraries

User-level thread systems are similar to conventional OS threads; both models allow applications to create multiple threads to expose tasks that can be executed simultaneously. During execution, threads can yield when they become idle and when thread execution ends, threads join with a parent thread. OS threads are typically scheduled preemptively where the OS periodically interrupts the execution of threads to perform a context switch that exchanges the currently executing thread for another thread. The preemptive scheduling model guarantees that all OS threads make forward progress. User-level threads, on the other hand, are scheduled cooperatively where a thread either completes or executes a yield operation to enable another thread to execute. The cooperative scheduling model relies on threads to cooperatively yield in order for other threads to make forward progress.

A user-level threading yield operation may automatically choose the next thread to execute. Alternatively, the yielding thread can supply a context to specify the next thread, as is done with the Unix* ucontext and Boost [10] fcontext APIs. In such models, thread contexts are continuations that capture the state of a suspended thread and these models are commonly used as a lower layer by user-level threading and multitasking systems.

User-level thread and task execution models have been explored extensively in the context of HPC programming [11]. The Argobots [32] threading package used in this work provides a lightweight, user-level threading model that is similar to that of QThreads [35], MassiveThreads [23], Intel® Thread Building Blocks [28], and Stack-Threads [34]. Such user-level threading systems transparently map user level threads to one or more underlying OS threads. These OS-level threads provide the execution resource on which user-level threads are executed and are often referred to as shepherd threads. The parallel execution of these shepherd threads are, in turn, managed by the OS. While this mapping does allow for over-subscription, in HPC workloads, shepherds are typically not oversubscribed and are pinned to a specific processor core.
2.4 Integrated User-Level Threading

Lightweight threading has been extensively explored in the context of MPI applications. MPC-MPI [25] and FG-MPI [19] have explored supporting multiple MPI processes as threads within a single OS process. To increase overlapping between communication and computation, MPI/SMPS [21] proposed a hybrid environment of MPI and task-based shared memory programming model [26] that allow the asynchronous communication among processes. Adaptive MPI [18] executes MPI processes as Charm++ tasks that can be adaptively scheduled. Castillo et al. [12,13] proposed to leverage MPI internal information to task-based runtime systems for making better scheduling decisions, whereas Sala et al. [29] utilized the external event information to pause and resume scheduled tasks. The most closely related research work to this work is MPI+ULT [20], which explored the usage of hybrid programming with MPI and user-level threads. As we explore in this work, the asynchronous, one-sided communication model provided by OpenSHMEM introduces unique challenges and opportunities to integration with user-level threading.

3 Enabling Thread Integration in OpenSHMEM

Supporting user-level threading effectively in an OpenSHMEM library requires integration of cooperative thread scheduling to prevent deadlock in scenarios where threads block or poll on updates to memory (e.g. in a call to shmem_wait or repeated calls to shmem_test) and to hide latency in scenarios where threads become blocked on communication (e.g. in a call to shmem_get). We propose several OpenSHMEM API extensions to register callbacks and provide query routines needed to support cooperative user-level thread scheduling in the OpenSHMEM library. The proposed API routines are shown in Listing 1 as OpenSHMEM extensions, prefixed by shmemx_.*.

Among the proposed APIs, the only routine required to support user-level threading is shmemx_register_yield, which registers a callback that the OpenSHMEM library invokes to yield the current thread. In our proposed API, the yield function is defined to take no arguments and utilize the scheduling policies as defined by the application.
To enable deep integration of user-level threading with OpenSHMEM, we define two registration routines that register callbacks enabling OpenSHMEM implementations to collect user-level thread specific information, such as thread ID, shepherd ID, and thread handle. These optional routines enable the OpenSHMEM library to track the runnable state of threads that block on communication and use this information to optimize thread scheduling. To enable this usage model, we introduce an initialization API for setting up the thread scheduler before creating the threads. Through this initialization routine, the user can optionally provide a configuration argument that specifies the total number of shepherd threads and user-level threads. This configuration setting can also allow the user to customize the scheduler, for example to prioritize threads performing one RMA operation to another to optimize forward progress of the threads. For example, if an application is executing two sets of threads with one set executing a fetch operation and the other executing a put using the fetched content, the user can set the fetch operation priority higher so that the scheduler prioritizes those threads leading to improved communication overlap. The shmemx_ult_scheduler_finalize routine releases all resources and resets all the counters associated with the scheduler.

The optional routine shmemx_get_nextRunnable_ult allows a user-level thread scheduler to query OpenSHMEM for a thread handle that is ready to be run. The library implementation can define its own policy on choosing the next runnable thread. The query routine shmemx_getRegistered_ult_count returns the total number of registered user-level threads within the OpenSHMEM layer. This information will provide application flexibility to choose the next thread either from the already executing ones or from the ones that have not been started yet, if any. Finally, we provide a routine that unregisters a user-level thread from the OpenSHMEM library, removing it from the internal data structure that holds the thread information. A code example demonstrating the API usage is shown in Listing 2.

Through the optional APIs listed in Listing 1, we provide the user flexibility to control the scheduling policy as needed. An alternative approach will be to design the thread scheduling transparent to the user, without providing any control. While such a design approach would greatly reduce the complexity of using the APIs, it would require the OpenSHMEM library either to communicate with the underlying thread library that the user chooses or to implement the desired thread library functionalities within itself.

4 Design of OpenSHMEM with Integrated User-Level Threading

An architectural view of user-level threading integrated with OpenSHMEM is shown in Fig. 2(a). In this model, the application uses a user-level threading package to parallelize the workload within an OpenSHMEM PE and individual threads perform OpenSHMEM operations. In the OpenSHMEM 1.4 API, all point-to-point communication operations are associated with an application-level context and users can assign threads to different contexts to enable communication isolation.

We have extended the open source Sandia OpenSHMEM (SOS) library to support the proposed model, namely with the ability to yield and transfer execution among the user-level threads through callbacks registered by the application. By default, SOS
invokes the registered user-level threading yield function to participate in cooperative scheduling. A more advanced communication-aware thread scheduling module is also provided that tracks the communication state of individual threads to identify runnable threads from within the OpenSHMEM layer and avoid the overhead of switching to threads that are still blocked on OpenSHMEM operations. This module utilizes completion counters \[14,27\] associated with the OpenSHMEM contexts used by individual threads to track completions and identify runnable threads.

To hide latency associated with blocked one-sided communication operations, we extend the communication flows used by SOS as shown in Fig. 2(b). This figure shows the existing sequence of operations used to implement OpenSHMEM operations using the OpenFabrics Interfaces libfabric communication layer. In the existing approach, an OpenSHMEM blocking RMA operation relies on waiting on any update on the event counters provided by the OFI through invoking fi_cntr_wait. With our proposed changes, an RMA operation initiated by a user thread checks for completion, releases any locks, and yields to the next runnable thread if there is no update on the event counter associated with the given context. In this way, another thread can commence its execution while the original thread goes to a pending state and returns when the blocking operation gets completed.

### 4.1 Implementation of Communication Aware Thread Scheduling

Upon initialization by the application, the scheduler allocates a queue to hold thread data objects for threads blocked on OpenSHMEM operations. The thread data object stores thread ID, corresponding shepherd ID, thread handle, etc. It also contains the
OpenSHMEM context that the thread is associated with along with the operation type (e.g. put, get) on which the thread is currently blocked. In addition, it maintains a flag indicating whether the thread is currently runnable. The queue data structure is used to support three basic operations as shown in Fig. 3(a):

**Append:** New threads are appended and remain in the queue until they are unregistered.

**Update:** Existing threads can be updated, e.g. when yielding in a blocking operation. An update operation moves the thread object to the end of the queue.

**Remove:** Upon completion, threads are removed from the queue. This is achieved at the application level by calling either `shmemx_ult_unregister` or `shmemx_ult_scheduler_finalize`.

After the scheduler is initialized, the library appends threads to the queue the first time a thread blocks on an RMA, AMO, or synchronization operation. Immediately
after issuing the operation, the thread checks whether it is complete by reading the event counter through `fi_cntr_read`. If the operation is not complete, it adds or updates its current status to the queue and returns control to the application through the `yield` callback. When new threads are pending, the yield routine performs a generic yield operation to start additional threads and maximize communication overlap. If no new threads are pending, the application-provided yield routine queries the communication-aware thread scheduler for a ready thread and yields to it by invoking the `yield_to` routine from the threading library. If neither case is met, the current thread continues execution. This execution flow is highlighted in Fig. 3(b).

We design the scheduler to dynamically detect the next runnable thread and provide the thread handler upon request through the `shmemx_get_nextRunnable_ult` API. To detect the next runnable thread, the scheduler leverages the completion tracking on individual OpenSHMEM contexts as described in [15]. In SOS, completion tracking is implemented by unsigned 64-bit integers that provide the number of issued and completed operations on a given context. These counters are used to identify whether a thread associated with a given context has made progress in their previously blocked operation. Listing 3 presents the pseudocode for identifying and obtaining the next runnable thread in our design.

As illustrated in Listing 3, we maintain a `nextRunnable` object to point to the next runnable thread. On each invocation of `shmemx_get_nextRunnable_ult`, we update the `nextRunnable` to point to the next thread in the queue that is runnable and the current `nextRunnable` is returned. If the number of runnable threads in the queue falls under a threshold `minRunnablesCount`, the queue is traversed to check all the thread contexts and the appropriate runnable threads are flagged. During this check, for each thread context, the issued and completed counters are read. The threads are marked runnable in the case of matching issued and completed counter values. If a particular operation is prioritized by the user during the scheduler initialization, the runnable threads are selected based on the operation type first and then the counter values. For simplicity, we present the scheduler algorithm without the priority based selection in Listing 3.

4.2 Simplifying Communication Overlap

Our proposed extensions presented in Sect. 4.1 enable an application to be re-written with user-level threads using only blocking communication APIs. This simplifies the way a user achieves communication overlap in an application that is otherwise implemented with non-blocking APIs. While we present the performance comparisons between these two executions in Sect. 5, we highlight the application-level code changes in this section through an example.

Listing 4 shows the key exchange phase in parallel integer sorting application, ISx [17], with a single thread and non-blocking APIs. To achieve overlapping between the two communication operations, `shmem_fetch_add_nbi` and `shmem_put_nbi`, the code is split into two separate loops. Before invoking a put operation, we wait for the corresponding destination offset fetch operation to be complete, thus overlapping the remaining fetch operations with the put operations. Achieving communication overlap in this way requires careful consideration and manual interleaving of the APIs from the application developer’s perspective.
Listing 3. Algorithm to detect and return the next runnable thread

```c
procedure get_next_runnable()
if next_runnable = NULL then
    if count_runnables \leq\ min_runnables_count and find_next_runnables() = 0 then
        return NULL
    end
end
ret_object ← next_runnable
update next_runnable with the next runnable thread in queue
return ret_object

procedure find_next_runnables()
    count ← 0, curr_thread ← thread_queue
    while curr_thread \neq NULL do
        if read_issued(curr_thread.ctx) = read_completed(curr_thread.ctx) then
            curr_thread.is_runnable ← true, count ← count + 1
        if next_runnable = NULL then
            next_runnable ← curr_thread
        end
        curr_thread ← curr_thread.next
    done
return count
```

Listing 4. Key exchange in ISx with single thread and non-blocking APIs

```c
static long long bucket_offset = 0;
static int bucket_keys[BUCKET_SIZE];
...
long long dest_offsets = malloc(sizeof(long long) * shmem_n_pes());
for (int i = 0; i < shmem_n_pes(); i++) dest_offsets[i] = -1;
for (int i = 0; i < shmem_n_pes(); i++) {
    int dest_pe = peers_iter(i);
    shmem_longlong_atomic_fetch_add_nbi(&bucket_offset, bucket_sizes[dest_pe],
                                        &dest_offsets[dest_pe], dest_pe);
}
for (int i = 0; i < shmem_n_pes(); i++) {
    int dest_pe = peers_iter(i);
    shmem_longlong_wait_until(&dest_offsets[dest_pe], SHMEM_CMP_NE, -1);
    shmem_int_put_nbi(&bucket_keys[dest_offsets[dest_pe]], ..., dest_pe);
}
shmem_quiet();
```

On the other hand, our proposed extensions enable OpenSHMEM to efficiently schedule user-level threads that allow the same key exchange program to be re-written as shown in Listing 5. In this version, we use the blocking communication APIs, and the overlapping among different operations is achieved through the usage of user-level threads and communication-aware thread scheduling provided by the underlying implementation. As illustrated in Listing 5, this approach achieves operation overlapping from a single loop execution with load balanced across multiple threads using separate contexts, which reduces the level of code complexity.
5 Experimental Results

We have extended Sandia OpenSHMEM (SOS) v1.4.4 with support for user-level threading integration. SOS was built using libfabric version 1.7.0 with the PSM2 [4] provider. SOS is configured with manual progress enabled with a progress interval of 1 us. We also disable bounce buffering to ensure consistent performance across different message sizes. We use the MPICH Hydra process launcher version 3.2 to execute all jobs and restrict processes to be bound to two CPU cores (--bind-to=core:2).

For the user-level thread library, we use Argobots [3,32] throughout our experiments. In our experiments, we analyze performance by utilizing both user-level threads and shepherd threads. A similar but alternative evaluation strategy would be to evaluate our proposed OpenSHMEM extensions in conjunction with BOLT [9], an OpenMP [6] based parallel library that utilizes Argobots for implementing the underlying threading mechanisms. As we provide the preliminary study on OpenSHMEM with user-level threads in this work, we plan to investigate an integration with other threading models in the future.

Results were gathered on a cluster with 8 compute nodes. Each compute node contains two Intel® Xeon® Platinum 8170 (Skylake) CPUs at 2.1 GHz and 192 GB of DDR4-2666 RAM. Each node contains one 100 Gbps Intel® Omni-Path Host Fabric Adapter 100 Series (Intel® OPA) and nodes are connected using an Intel® Omni-Path Edge Switch. Nodes are running Red Hat® Enterprise Linux Server release 7.5 (Maipo) with Linux® kernel 3.10.0-862.el7.x86_64.

5.1 Performance Analysis of Different Scheduler Policies

We first analyze the performance impact of different thread scheduling policies. We use either the round-robin (RR) or random policy for the Argobots thread scheduler to schedule uninitialized threads. Once registered with SOS, threads are scheduled by the integrated thread scheduler using a round-robin (RR), random, or communication aware policy. We conduct these experiments on 4 nodes with 4 PEs per node using the Key
Exchange pattern introduced in Sect. 5.2, which performs an atomic fetch-add followed by a put operation. We evaluate cases where the workload is balanced and unbalanced across threads. Imbalance is introduced by creating additional threads that wait for and consume data as it arrives.

Figure 4 presents the results of these experiments, where the legend “A”+“B” indicates that thread library scheduler with policy “A” and OpenSHMEM thread scheduler with policy “B” is used. We present only one instance of the random scheduling for uninitialized threads used by the thread library scheduler as it has similar performance impact to the RR scheduling policy.

For balanced load experiments with 16 threads per PE presented in Fig. 4(a), we observe that our proposed communication-aware thread scheduler increases overhead, resulting in a 3–8% increase in latency compared to the default round-robin policy for message sizes up to 256 B. Larger message sizes incur higher latency, thus increasing the opportunity for communication-aware scheduling and achieving 3–5% performance improvement compared to the default round-robin policy. For the unbalanced load distribution with 64 threads per PE shown in Fig. 4(b), communication-aware thread scheduling uses the internal communication state to avoid scheduling blocked threads, improving performance by 23% across most message sizes. In both of these cases, random scheduling performs poorly because it ignores the current communication state, causing it to frequently select blocked threads for execution.

5.2 Micro-benchmark Case Studies

We identify several communication patterns that are commonly used in OpenSHMEM applications and create micro-benchmarks to analyze their performance with user-level threading. Our micro-benchmarks support both blocking and nonblocking communication and can be run with or without user-level threads. The resulting case studies provide a base case of potential communication performance improvement with user-level threading. A key area of inquiry is whether user-level threading can achieve communication performance similar to that of nonblocking communication, but with lower code complexity. We conduct each of these experiments on 4 nodes with 16 PEs per node. Latency is reported averaging 1000 iterations for each message size. For multi-threaded experiments, we run with one shepherd and 2, 8, or 32 user-level threads.

Streaming: This micro-benchmark performs a unidirectional streaming bandwidth test where a group of sender PEs send data to a group of receiver PEs using shmem_put
operations. Figure 5 show the bandwidth with 8 senders and 8 receiver PEs on two nodes. For cases with threads, we vary the number of user-level threads while keeping a single shepherd thread.

With blocking communication, shown in Fig. 5(a), we observe that user-level threads can improve the achieved bandwidth compared to that of single-threaded implementation. For 32 B–1 KB message size, 32 user-level threads provide 2.22x–2.64x more bandwidth compared to single-threaded execution. However, with non-blocking APIs, as presented in Fig. 5(b), user-level threads do not bring any additional benefits. Since this benchmark has only one operation, using non-blocking API for that operation yields the same performance between single and multiple user-level threads.

**Transpose:** In this study, we consider an all-to-all communication operation using \texttt{shmem\_put}. Each PE runs a loop of all the PEs and use the loop index to construct the destination PE, sending different data to different PEs. This communication pattern represents a generic use-case in OpenSHMEM applications. For example, distributed matrix transpose, OpenSHMEM implementation of LAMMPS application [33], and distributed fast Fourier transform may utilize this communication pattern. Listing 6 presents this communication loop example with blocking API.

We develop a micro-benchmark for the communication pattern shown in Listing 6. When nonblocking \texttt{shmem\_int\_put\_nbi} operations are used, we place a \texttt{shmem\_quiet} after the loop to ensure completion of all the pending put operations. For multithreaded execution, we divide the loop across threads to assign each thread a set of destination PEs for the communication. We use separate contexts for each thread to avoid synchronization between threads. Figure 6 presents the results obtained from this experiment for up to 4 KB message size.

As shown in Fig. 6(a), user-level threading reduces latency with blocking communication by almost 42% with 8 threads for most message sizes (4 B–2 KB). In Fig. 6(b), we observe that with nonblocking communication user-level threads do not improve
latency for sizes up to 32 B. For message sizes larger than 32 B, we observe a maximum performance improvement of 20% (for 2 KB message size) using 8 threads. In contrast to the blocking API, we observe performance degradation with 32 user-level threads for message sizes up to 128 B.

**Key exchange:** In this case study, we explore a communication pattern that is common to the key exchange phase of parallel sorting, similar to the pattern used by the Integer Sort (ISx) [17] benchmark. This communication pattern involves an atomic fetch-add operation followed by a put operation utilizing the fetched value. This pattern is used when different PEs append data to the same destination buffer on a remote PE and an atomic `fetch_add` to reserve buffer space at the destination PE. Listing 7 highlights this communication loop block with blocking APIs for `shmem_fetch_add` and `shmem_put`. For the non-blocking implementation, `shmem_fetch_add_nbi` and `shmem_put_nbi` are split into separate loops and a `shmem_quiet` operation is used to complete operations after each loop, as shown in Listing 4.

As shown in Fig. 7(a), we observe 46–52% performance improvement for message sizes up to 64 B and a maximum of 40% improvement for larger message sizes. With non-blocking APIs shown in Fig. 7(b), we observe a maximum of 27% improvement for 2 KB message size with 8 threads.

**Put with Signal:** Listing 8 shows the blocking version of commonly used OpenSHMEM communication loop that performs an all-to-all exchange where each iteration sends a message and then sets a signal flag at a given peer PE to notify that the data has arrived. For the non-blocking API, we use the nonblocking `shmem_put_signal_nbi` routine that performs the data transfer and subsequent signal flag update as a single operation. Put-with-signal has been ratified for OpenSHMEM 1.5 and is available in SOS.

---

**Listing 7. Case study 3 - All-to-all with shmem_fetch and shmem_put**

```c
for (int i = 0; i < shmem_n_pes(); i++) {
    int dest_pe = peers_iter(i);
    long long dest_offset = shmem_longlong_atomic_fetch_add(&offset, ..., dest_pe);
    shmem_int_put(&dest_buffer[dest_offset], ..., dest_pe);
}
```
As shown in Fig. 8(a) and Fig. 8(b), we observe more than 40% performance benefit for both blocking and nonblocking APIs with the addition of user-level threads across all message sizes.

5.3 Application Case Studies

We analyze the application-level performance impact of integrated user-level thread scheduling using three benchmarks: Mandelbrot set generation, integer sort for exascale (ISx), and Smith-Waterman DNA sequence alignment.

Mandelbrot: We use the OpenSHMEM implementation [2] of Mandelbrot set generator provided through the SOS repository and first introduced in [14]. We conduct two sets of experiments with this application. In the first set, we vary the total number of user-level threads per PE while keeping two shepherd threads for both default and modified implementation. We measure the speedup obtained with respect to total work rate compared to the default threaded implementation. We conduct this experiment on 8 nodes with 16 PEs per node. We use 8 K as the width and height of the Mandelbrot domain. We vary the number of user-level threads from 1 per PE (32 per node) to 64 per PE (2K per node). We compare the performance of user-level threading with two shepherds (2 pthreads) for the default implementation. Both cases use the same number of cores without OS thread oversubscription. We measure performance for three different communication variants provided by the benchmark: Blocking, Non-blocking, and Non-blocking pipelined (shown as NB-pipelined).

As shown in Fig. 9(a), with 16 user-level threads per PE, we observe 1.35× speedup for Blocking, 5% improvement for Non-blocking, and 1.57× speedup for Non-blocking with pipelining. These results demonstrate that the introduction of user-level threads can provide significant performance improvements without OS thread oversubscription. We further analyze the performance in the presence of OS thread oversubscription by using
8 pthreads and shepherds threads for both the default and modified implementation of Mandelbrot, respectively. We use 8 user-level threads in this experiment and keep other settings same as the previous experiment. In Fig. 9(b), we report the total work rate and compare this between the two implementations. We observe similar performance benefits for all three settings: 1.21× for Blocking, 5% for Non-blocking, and 1.94× for Non-blocking with pipelined communication using contexts.

**ISx:** We conduct a weak scaling experiment with the Integer Sort (ISx) [17] benchmark on 8 nodes. As the current ISx implementation is single-threaded, we conduct this experiment with one shepherd thread and 64 user-level threads. We vary the total number of PEs from 4 per node to 16 per node with a fixed 64M keys per PE. We compare the performance with respect to average all-to-all time per PE reported by the benchmark. We measure this performance with the default setting of 1 warm-up and 1 test iteration. As shown in Fig. 10(a), we observe a maximum of 14.7% performance benefit for 96 PEs compared to the default non-threaded implementation.

**Smith-Waterman:** Smith-Waterman is a dynamic programming algorithm used for matching similarity between two DNA/RNA sequence, which locates regions in sequence with high levels of similarity. The OpenSHMEM implementation of this algorithm is first proposed in [7] and its open-source implementation is available in [5]. We observe that the OpenSHMEM Smith-Waterman benchmark performs a large number of RMA operations with complex interactions between communication and computation.
phases. Thus, we anticipate that user-level threads can provide a productive method for hiding communication latency and overlapping communication with computation.

Figure 10(b) shows performance across different scales for four different settings based on API and pre-fetching options: the default implementation (Default), default with pre-fetching enabled and non-blocking APIs (Default-Prefetch-NB), enhanced implementation with user-level threads (User-level-threads), and user-level threads with non-blocking APIs (User-level-threads-NB). We conduct this experiment on 8 nodes with 4 PEs per node and 8 user-level threads per PE. We measure the time taken on kernel 1 execution of the implementation and compare the performance between the baseline user-level threaded versions. We observe that with user-level thread scheduling, performance of the algorithm improves by almost 28.74% compared to the baseline. With pre-fetching and non-blocking APIs, the algorithm performs slightly better compared to the user-level threaded implementation with blocking APIs. However, with non-blocking APIs and even without pre-fetching, user-level threaded implementation can out-perform the default best case by 1–3%.

To illustrate the additional overlap introduced by user-level threads, we utilize the performance counters [27] in SOS and analyze the number of pending communication operations for Smith-Waterman implementation. We conduct this experiment on 4 nodes with 4 PEs per node with a scale value of 25 and observe the differences between the default execution and user-level threaded execution. As presented in Fig. 11, user-level threads introduce better overlapping (increased number of pending operations in Fig. 11(b)) and thus, reduces the execution time.

Fig. 10. Performance improvement of ISx and Smith-Waterman algorithm with user-level threads

Fig. 11. Overlapping through performance counters for Smith-Waterman algorithm
6 Conclusion

This paper explores the usage of user user-level threading with OpenSHMEM as an effective method of exposing communication overlap, while maintaining the ease-of-programming provided by blocking communication interfaces. We propose a generic OpenSHMEM API extension to enable cooperatively scheduled threads to safely use blocking OpenSHMEM interfaces. We further build on these concepts to introduce communication-aware thread scheduling for OpenSHMEM applications that leverages the OpenSHMEM runtime system’s knowledge of multithreaded communication state to avoid scheduling blocked threads, thereby minimizing overheads.

Our experimental analysis indicates that user-level threading is effective at enabling communication overlap and pipelining. Microbenchmark results showed that blocking communication with user-level threading can provide performance comparable to optimized, single-threaded nonblocking communication. For example, in a majority of cases analyzed in Sect. 5.2, we observe that the blocking API implementation with user-level threads meets or exceeds the performance of single-threaded non-blocking implementations for message sizes larger than 128 B. Similar results were observed with the Mandelbrot and Smith-Waterman benchmarks presented in Figs. 9(a) and 10(b), respectively. We attribute overheads at smaller message sizes to threading inefficiencies that can be addressed with greater attention to threading support in the communication stack.

In this work, our proposed OpenSHMEM extensions define a generic infrastructure for building communication-aware schedulers. While the scheduler we have demonstrated is effective, this remains a broad area for further investigation and customization. Also, the usage of user-level threading in conjunction with the new OpenSHMEM features, such as the proposed teams interface, may provide new opportunities for performance optimization.

Intel and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as “Spectre” and “Meltdown”. Implementation of these updates may make these results inapplicable to your device or system.

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/benchmarks.

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

1. Complete Context Control. https://www.gnu.org/software/libc/manual/html_node/System-V-contexts.htm
2. Mandelbrot in Sandia OpenSHMEM. https://github.com/Sandia-OpenSHMEM/SOS/blob/master/test/apps/mandelbrot.c
3. Official Argobots Repository. https://github.com/pmodels/argobots
4. OPA-PSM2. https://github.com/intel/opa-psm2
5. Smith-Waterman algorithm in SSCA1. https://github.com/ornl-languages/osb/tree/master/ssca1
6. The OpenMP API Specification. https://www.openmp.org/
7. Baker, M., Welch, A., Gorentla Venkata, M.: Parallelizing the smith-waterman algorithm using OpenSHMEM and MPI-3 one-sided interfaces. In: OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies, pp. 178–191 (2015)
8. Bendersky, E.: Measuring context switching and memory overheads for Linux threads. https://github.com/eliben/code-for-blog/tree/master/2018/threadoverhead
9. Bolt is openmp over light-weight threads. https://www.bolt-omp.org/
10. Boost c++ libraries. https://www.boost.org
11. Castelló, A., Peña, A.J., Seo, S., Mayo, R., Balaji, P., Quintana-Ortí, E.S.: A review of lightweight thread approaches for high performance computing. In: IEEE International Conference on Cluster Computing (CLUSTER), pp. 471–480, September 2016
12. Castillo, E., et al.: Optimizing computation-communication overlap in asynchronous task-based programs. In: Eigenmann, R., Ding, C., McKee, S.A. (eds.) Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, 26–28 June 2019, pp. 380–391 (2019)
13. Castillo, E., t al.: Optimizing computation-communication overlap in asynchronous task-based programs. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, p. 415–416. PPoPP 19, New York, NY, USA (2019)
14. 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. New York, NY, USA (2014)
15. Grossman, M., Doyle, J., Dinan, J., Pritchard, H., Seager, K., Sarkar, V.: Implementation and evaluation of OpenSHMEM contexts using OFI libfabric. In: Gorentla Venkata, M., Imam, N., Pophale, S. (eds.) OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence, pp. 19–34. Cham (2018)
16. Grun, P., et al.: A brief introduction to the OpenFabrics interfaces - a new network API for maximizing high performance application efficiency. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 34–39, August 2015
17. Hanebutte, U., Hemstad, J.: ISx: a scalable integer sort for co-design in the exascale era. In: 9th International Conference on Partitioned Global Address Space Programming Models, pp. 102–104, September 2015
18. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Rauchwerger, L.(ed.) Languages and Compilers for Parallel Computing, pp. 306–322 (2004)
19. Kamal, H., Wagner, A.: FG-MPI: fine-grain MPI for multicore and clusters. In: IEEE International Symposium on Parallel Distributed Processing, Workshops and Ph.d. Forum (IPDPSW), pp. 1–8, April 2010
20. Lu, H., Seo, S., Balaji, P.: MPI-ULT: overlapping communication and computation with user-level threads. In: IEEE 17th International Conference on High Performance Computing and Communications, pp. 444–454, August 2015
21. Marjanović, V., Labarta, J., Ayguadé, E., Valero, M.: Overlapping communication and computation by using a hybrid MPI/SMPSS approach. In: Proceedings of the 24th ACM International Conference on Supercomputing, pp. 5–16. ICS 2010, NY, USA (2010)
22. MPI Forum: MPI: a message-passing interface standard version 3.1. Technical report, University of Tennessee, Knoxville, June 2015
23. Nakashima, J., Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, pp. 222–238. Heidelberg (2014)
24. OpenSHMEM application programming interface, version 1.4. http://www.openshmem.org, December 2017
25. Pérache, M., Jourdren, H., Namyst, R.: MPC: a unified parallel runtime for clusters of NUMA machines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008 - Parallel Processing, pp. 78–88. Heidelberg (2008)
26. Perez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: 2008 IEEE International Conference on Cluster Computing, pp. 142–151, September 2008
27. Rahman, M.W.U., Ozog, D., Dinan, J.: Lightweight instrumentation and analysis using OpenSHMEM performance counters. In: OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity. pp. 180–201 (2019)
28. Reinders, J.: Intel Threading Building Blocks. First edn, Sebastopol, CA, USA (2007)
29. Sala, K., et al.: Improving the interoperability between MPI and task-based programming models. In: Proceedings of the 25th European MPI Users Group Meeting. EuroMPI18, New York, NY, USA (2018)
30. Sandia OpenSHMEM (2018). https://github.com/Sandia-OpenSHMEM/SOS
31. 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 and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments, pp. 97–113. Cham (2016)
32. Seo, S., et al.: Argobots: a lightweight low-level threading and tasking framework. IEEE Trans. Parallel Distrib. Syst. 29(3), 512–526 (2018)
33. Tang, C., Bouteiller, A., Herault, T., Gorentla Venkata, M., Bosilca, G.: From MPI to OpenSHMEM: porting LAMMPS. In: OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies, pp. 121–137 (2015)
34. Taura, K., Tabata, K., Yonezawa, A.: Stackthreads/mp: Integrating futures into calling standards. In: ACM SIGPLAN Symposium Principles Practice Parallel Program (1999)
35. Wheeler, K.B., Murphy, R.C., Thain, D.: Qthreads: an API for programming with millions of lightweight threads. In: IEEE International Symposium on Parallel and Distributed Processing, pp. 1–8, April 2008
