Reducing Synchronization Overhead Through Bundled Communication

James Dinan, Clement Cole, Gabriele Jost, Stan Smith, Keith Underwood, and Robert W. Wisniewski

Intel Corp.
{james.dinan,clement.t.cole,gabriele.jost,stan.smith,
keith.d.underwood,robert.w.wisniewski}@intel.com

Abstract. OpenSHMEM provides a one-sided communication interface that allows for asynchronous, one-sided communication operations on data stored in a partitioned global address space. While communication in this model is efficient, synchronizations must currently be achieved through collective barriers or one-sided updates of sentinel locations in the global address space. These synchronization mechanisms can oversynchronize, or require additional communication operations, respectively, leading to high overheads. We propose a SHMEM extension that utilizes capabilities present in most high performance interconnects (e.g. communication events) to bundle synchronization information together with communication operations. Using this approach, we improve pingpong latency for small messages by a factor of two, and demonstrate significant improvement to synchronization-heavy communication patterns, including all-to-all and pipelined parallel stencil communication.

SHMEM is a popular partitioned global address space (PGAS) parallel programming model, and it has been in use for over two decades [4]. Recently, the SHMEM library has been codified as an open, community standard in the OpenSHMEM 1.0 specification [19]. SHMEM provides a global address space that spans the memory of the system and allows the programmer to create symmetric objects, which are present at all processing elements (PEs). These objects can be read and updated using one-sided get and put operations.

While SHMEM provides high-performance one-sided data movement operations, it includes few primitives for synchronizing between PEs. In the current Open SHMEM standard, synchronization can be achieved using a collective barrier, or by polling or waiting on a flag that will be remotely updated using a one-sided operation. While these synchronization primitives are sufficient for achieving point-to-point and global synchronization, they are not able to fully utilize capabilities provided by modern high performance interconnects. In particular, barrier synchronization can generate more synchronization than is needed by the algorithm, and its performance can be negatively impacted by system noise and imbalance. In addition, point-to-point synchronization using counting or boolean flag locations in the global address space requires additional communication when updating the flag.

S. Poole, O. Hernandez, and P. Shamis (Eds.): OpenSHMEM 2014, LNCS 8356, pp. 163–177, 2014.

Modern networks used in high performance computing systems provide a variety of mechanisms that can be used to bundle synchronization and communication. One such example is communication events, which notify the recipient that a one-sided communication operation has arrived and is complete.

We present a new synchronization extension to OpenSHMEM, called counting puts, that utilizes network-level events to provide efficient point-to-point synchronization. Counting puts can utilize communication completion events to inform a PE that it has been the target of a one-sided communication operation, and that the data written is available to read. Counting puts effectively enables receiver-side synchronization. In contrast, existing point-to-point synchronization in SHMEM is sender-side, and requires additional communication to update flag locations at the target PE.

We describe the counting puts interface and its implementation in an open source SHMEM implementation for the low-level Portals networking API [6]. We demonstrate that bundling communication for a Portals network reduces communication latency by half for small messages, and that it significantly improves the bandwidth achieved to the synchronization-heavy all-to-all communication algorithm. We further demonstrate the performance impact of counting puts on a pipelined parallel stencil computation, that relies heavily on point-to-point synchronization. While our evaluation focuses on a Portals implementation, we describe several methods for creating efficient receiver-side implementations of counting puts that can achieve the demonstrated performance improvements on a variety of networks.

1 Overview

The SHMEM parallel programming model provides a global address space, shown in Figure 1, where the memory of each processing element (PE), or SHMEM process, is partitioned into private and shared segments. Data in the private segment is accessible locally, while data in the shared segment is accessible both locally and remotely, through SHMEM library routines. The shared segment contains both a shared heap, for dynamically allocated shared objects, and a shared data segment, which allows statically declared objects to be accessed by remote PEs.

Objects in a shared segment are symmetric, meaning that an instance of the object is accessible at every PE, and that the object can be accessed using the address of the corresponding symmetric object in the local PE's address space. Thus, when accessing data in the global address space, the target address is the pair containing the destination PE rank and the symmetric address. Remote accesses are performed using one-sided *get* and *put* data copy operations, that transfer data between local and remote buffers. In addition SHMEM provides a variety of collective and atomic, one-sided communication routines.

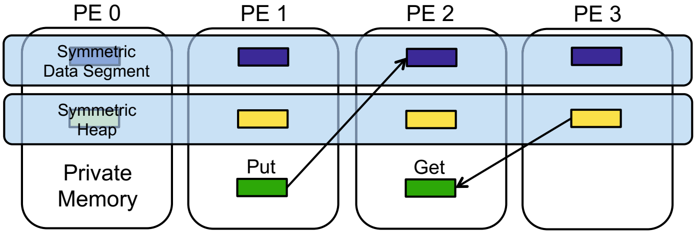

Fig. 1. SHMEM communication model, showing shared and private memory areas, and one-sided get and put communication operations

1.1 Portals

In this work, we demonstrate the counting puts interface using the open source OpenSHMEM implementation for the low-level Portals networking API [1,5]. 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. As data arrives at the target, acknowledgement messages are returned to the sender. Thus, when a process waits for communication operations to complete, it waits for acknowledgement messages from the target.

1.2 Synchronization in OpenSHMEM

The OpenSHMEM standard provides both collective and point-to-point synchronization primitives. SHMEM barriers are collective synchronization operations that can include all PEs, or a regular subset that includes PEs whose ranks are a multiple of a power of two. In addition to synchronizing the involved PEs, before returning, SHMEM barriers also ensure that all preceding writes to symmetric objects have completed.

Point-to-point synchronization is achieved through symmetric flag variables that are acted upon using one-sided operations. These flags can be updated using one-sided writes, when a single PE updates the flag, or atomic operations, when multiple PEs update the flag. A PE can wait for the value of the flag to satisfy a certain condition by using one of the waiting routines provided in the OpenSHMEM API. In addition, some applications also poll flag locations directly, which can require the use of additional system-specific memory fences to ensure data consistency.

```
for (pe = 0; pe < NPES; pe++)
    shmem_putmem(&data_recv[pe], &data_send[pe], data_size, pe);

shmem_barrier_all();
```

Listing 1.1. All-to-all with barrier synchronization

```
for (pe = 0; pe < NPES; pe++)
    shmem_putmem(&data_recv[pe], &data_send[pe], data_size, pe);

shmem_fence();

for (pe = 0; pe < NPES; pe++)
    shmem_int_add(&flag, -1, pe);

shmem_int_wait_until(&flag, SHMEM_CMP_EQ, 0);
```

Listing 1.2. All-to-all with point-to-point synchronization

For point-to-point synchronization, data consistency is achieved using either *fence* or *quiet* operations. A SHMEM fence operation provides ordering, by ensuring that any operations performed by the calling PE to a particular remote PE will be completed before any subsequent operations issued by the calling PE to the same remote PE. A SHMEM quiet operation provides a stronger ordering semantic, and ensures that all put operations performed by the calling PE will be remotely completed and visible to all PEs when the call to quiet returns

We illustrate these two synchronization mechanisms with a simple all-to-all communication example, similar to the type of communication that is performed in a fast Fourier transform or parallel sort. In this data exchange, each PE must wait until all data has arrived before proceeding with the next phase of the computation. Listing 1.1 shows this communication pattern when a barrier synchronization is used, and Listing 1.2 shows this communication pattern when point-to-point synchronization is used.

In comparison with barrier synchronization, point-to-point synchronization can be performed more efficiently, because a PE does not need to wait for all other PEs to receive data. However, the flag update operation requires an additional communication and, depending on the underlying network, a fence or quiet operation can require additional communication to ensure ordering or remote completion, respectively. Overhead from these operations can outweigh the benefits from relaxed synchronization. For example, on ordered networks, a fence is a no-op, but a quiet operation requires sending a round-trip message to all other PEs that the calling PE has communicated with, to flush the ordered communication channels. In comparison, on unordered networks, both fence and quiet operations require waiting for point-to-point remote completion with each other PE. In the case of a fence, the calling PE must only wait before performing additional communication operations, whereas a quiet requires waiting for communication with all other PEs to complete before returning. When using the Portals communication API, communication operations are completed by

```
void shmem_ct_create(shmem_ct_t *ct);
void shmem_ct_free(shmem_ct_t *ct);
long shmem_ct_get(shmem_ct_t ct);
void shmem_ct_set(shmem_ct_t ct, long value);
void shmem_ct_wait(shmem_ct_t ct, long wait_for);
void shmem_putmem_ct(shmem_ct_t ct, void *trg, void *src, size_t bytes, int pe);
```

Listing 1.3. Counting puts API extension

waiting for acknowledgement messages to be returned from the target PE to the source PE for each operation.

2 Bundling Communication and Synchronization

We propose an extension to OpenSHMEM that bundles communication and synchronization. A variety of bundled communication interfaces are possible, based on the operations that will be bundled and the interface that will be used to access notification information. For example, the ARMCI one-sided communication library [18] provides the ARMCI_Put_flag operation that bundles two put operations, where a notification flag in the target process' address space is updated after the main data payload has been delivered. While it is easy to use, this interface requires distinct flags for each PE that will perform a put-and-notify operation. For all-to-all communication, this can result in O(NPEs) flags at every PE.

Our proposed interface is shown in Listing 1.3. This interface bundles an atomic increment operation with the communication operation, allowing the flag to be shared by multiple communicate-and-notify operations. We use an opaque shmem_ct_t representation for the counter, to enable a broader variety of efficient implementations. When network events are used to signal completions, the SHMEM implementation or networking layer can locally increment the counter as operations are performed, rather than requiring the remote PE to perform the update. Our discussion of this interface focuses on providing support for a put-and-notify operation; however, the proposed interface and implementation can also be utilized to provide a get-and-notify operation to support producer-consumer computational patterns.

The proposed interface provides functions that can be used to create and free an event counter (CT); get and set the counter's value; and wait for the counter to reach a particular value [20]. New communication functions are also provided that add a CT parameter that should be updated when the operation has completed. An example all-to-all communication using the new interface is shown in Listing 1.4. In comparison with Listing 1.1, this example achieves point-to-point synchronization, and in comparison with Listing 1.2, this example can eliminate overheads associated with the fence operation and flag updates.

```
shmem_ct_create(&ct);

for (pe = 0; pe < NPES; pe++)
    shmem_putmem_ct(ct, &data_recv[pe], &data_send[pe], data_size, pe);

shmem_ct_wait(ct, NPES);
```

Listing 1.4. All-to-all with counting puts synchronization

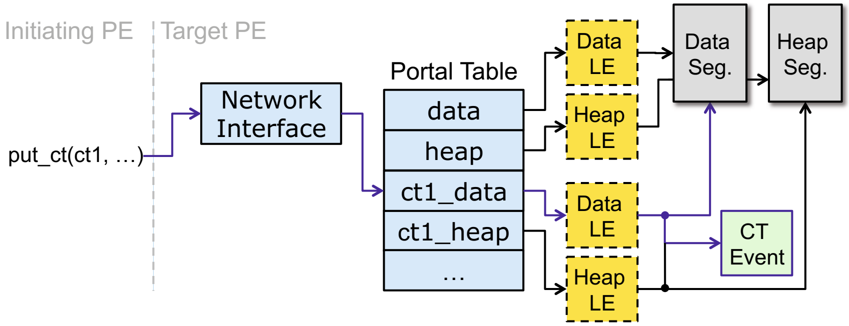

Fig. 2. Portals implementation of the counting puts interface

2.1 Implementation of the Counting Puts Interface

A variety of implementation strategies are possible for the counting puts interface; we chose an opaque representation of the CT object to provide more flexibility in the implementation. For example, in addition to enabling implementations that use low-level network counting events, the opaque CT object enables an implementation on top of the existing SHMEM interface. In such an implementation, a symmetric counter location is allocated at each PE during CT creation, and counted put operations perform a put, fence, and atomic increment using the functions provided in the SHMEM API.

Most networks provide mechanisms that can be utilized to implement these operations more efficiently by bundling communication and synchronization. On networks that are programmable, or on-load communication to the processor (e.g. sockets or PSM), CT information can be embedded in message headers enabling a receiver-side implementation to perform bookkeeping. Many networks report low-level communication events when one-sided operations complete in a given PE's memory. As these events are consumed in the SHMEM runtime system, the corresponding counter can be incremented.

We implement the CT interface in the open source Portals 4 [6] implementation of SHMEM [5]. We utilize Portals 4 counting communication events to achieve an efficient receiver-side implementation of counting puts. A high-level schematic of our implementation is shown in Figure 2. This example shows the Portals objects that are components in the implementation of the CT interface, and the flow of control when processing a counted put operation.

As shown in Figure 2 our implementation utilizes Portals lightweight counting events, that are incremented when each counted operation is completed at the target PE. Full events utilizing an event queue that can be attached to each portal table entry (not shown in Figure 2, for clarity) can also be used, resulting in an implementation similar to one that uses InfiniBand event queues. In such an implementation, CT query and wait routines would need to search this queue for operations affecting the counter. Counting events provide a more efficient implementation vehicle, as they use a fixed amount of memory and do not incur queue processing overheads.

Individual event counters are distinguished using distinct portal table entries that act as separate communication contexts. Communication operations in Portals specify the target network interface, portal table entry, and offset relative to the beginning of the memory portal. When the system can provide identical segment base addresses, a single portal table entry can be used to expose memory for one-sided access. On most clusters, separate portal table entries must be created for the static data and dynamic heap segments, because these segments can be disjoint in memory and located at different starting addresses across PEs. Prior to performing communication, the corresponding portal table entry is identified by comparing the symmetric address with the local addresses of the heap and data segments. The symmetric address is then converted to an offset relative to the beginning of the corresponding memory segment, and the offset and portal table entry are passed to the desired communication routine. Distinct heap and data segment portal table entries are created for each counter, allowing the implementation to identify which counter should be incremented when a counted put arrives. Non-matching list entries that describe the complete heap and data memory segments are attached to the respective portal table entries, and a counting event is registered with each list entry.

We note that the proposed interface does not guarantee any ordering or consistency beyond the completion of the counted communication operation. This targeted completion rule allows for greater concurrency and performance potential. Thus, we do not enforce any additional ordering in our implementation. If the algorithm requires ordering of non-counted operations, or ordering across different counted operations targeting the same PE, existing OpenSHMEM synchronization operations must be used.

3 Experimental Evaluation

We extended the Portals OpenSHMEM implementation [5] with the counting puts interface, and utilize the Portals 4 InfiniBand reference implementation [2] to provide Portals support. While this is not a native implementation of the Portals interface, it utilizes InfiniBand network events to implement Portals counting events, which allows us to demonstrate the relative performance improvement of the receiver-side counting puts protocol. This protocol eliminates additional messages that are generated when synchronizing through shared flag variables.

We utilize a 15-node cluster with a Mellanox QDR InfiniBand interconnect for experimentation. 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. We demonstrate the impact of the proposed CT interface on communication efficiency using ping-pong and all-to-all microbenchmarks. In addition, we demonstrate significant performance improvement for a pipelined parallel stencil computation, that relies heavily on point-to-point synchronization.

3.1 Ping-Pong Latency

We measured ping-pong latency using a simple benchmark with two PEs. In each iteration of the benchmark, one PE is the sender and one is the receiver. After each iteration, sender and receiver roles are reversed. For the baseline implementation using the operations available in the current OpenSHMEM specification, the sender performs the following sequence of operations.

```
shmem_putmem(rcv_buf, snd_buf, msg_length, target);
shmem_fence();
shmem_int_inc(&flag, target);
```

The receiver performs the following sequence of operations.

```
shmem_int_wait(&flag, 0);
flag = 0;
```

For the CT implementation of the benchmark, the sender performs the following operation.

```
shmem_putmem_ct(ct, rcv_buf, snd_buf, msg_length, target);
```

The receiver performs the following sequence of operations.

```
shmem_ct_wait(ct, 1);
shmem_ct_set(ct, 0);
```

The half round-trip latency is shown in Figure 3 for baseline and CT implementations. From this data, we see that the latency is approximately halved for small messages. For larger message sizes, the cost associated with the fence and flag update operations is amortized over a larger message transfer and results in a decreasing speedup from the CT extension.

3.2 All-to-All Bandwidth

We measure the bandwidth achieved using a simple all-to-all communication benchmark, where every PE sends a message to every other PE and waits for messages to arrive. For the baseline version of this benchmark, each PE performs the sequence of operations shown in Listing 1.5. For the CT version of the benchmark, the fence is omitted, and flags are replaced with a CT object, using the same approach as in the ping-pong algorithm. For the CT and flags versions of the benchmark, a pair of synchronization constructs is created and alternated across loop iterations to eliminate the race that arises in resetting the value of the counter or flag. In addition, a barrier synchronization version was created that replaces the fence and all flag operations with a single call to

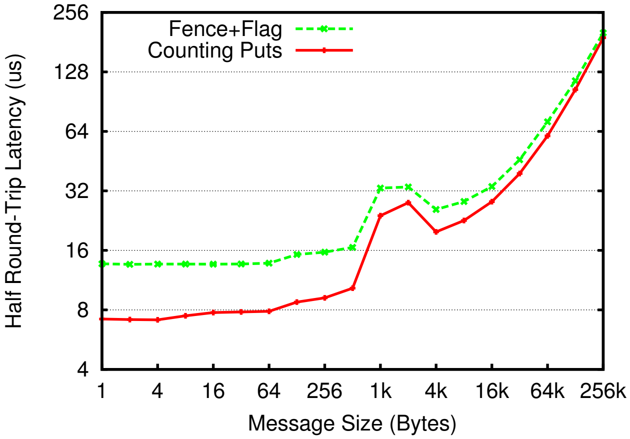

Fig. 3. Half round-trip latency for the ping-pong benchmark on the InfiniBand cluster

```
/* Initially, flag = num_pes */

pe = me;
do {
    shmem_putmem(&target_buf[me], &src_buf[pe], msg_size, pe);
    pe = (pe + 1) % num_pes;
} while (pe != me);

shmem_fence();

pe = me;
do {
    shmem_int_add(&flag, -1, pe);
    pe = (pe + 1) % num_pes;
} while (pe != me);

shmem_int_wait_until(&flag, SHMEM_CMP_EQ, 0);
```

Listing 1.5. Baseline implementation of the all-to-all communication benchmark

shmem_barrier_all(), which synchronizes all processes and ensures that all communication has been completed. Communication operations are staggered across PEs to spread out communication. While more sophisticated algorithms for all-to-all exist [10,21], this algorithm captures the approach that would be taken in a loosely synchronized or pipelined application.

The bandwidth achieved per node, when one PE is run per core, for each version of the all-to-all benchmark is shown in Figure 4. The barrier implementation achieves the lowest bandwidth because of the overhead from global synchronization. The fence implementation provides increased network efficiency, but incurs overhead from $O(\text{NPEs})$ additional communications per PE that are required to update the flag variables. By eliminating these operations, the CT version of the benchmark provides the best performance. As was the case with ping-pong latency,

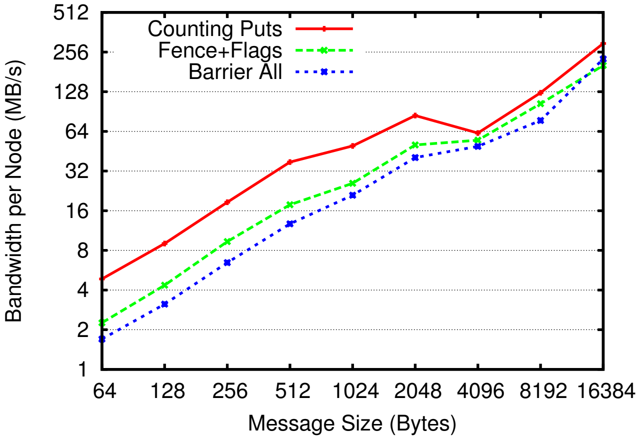

Fig. 4. All-to-all bandwidth achieved per node on the InfiniBand cluster

the cost of the additional synchronization communications is amortized over long transfer times, and the relative impact of increased communication is reduced.

The bandwidth we report in Figure 4 is significantly lower than the theoretical peak of 40 Gb/sec for our QDR InfiniBand network. This is caused by overhead incurred in simulating Portals communication on top of InfiniBand. However, these results still capture the performance improvement from eliminating additional messages needed to update flag locations at every PE.

3.3 Pipelined Parallel Stencil Kernel

Next, we investigate the performance impact of counting puts on a fine-grain pipelined parallel stencil computation. This type of computation has strong data dependencies across units of work, requiring frequent point-to-point synchronization. Pipelined parallel stencils are encountered in a variety of numerical methods, including the Lower-Upper Symmetric Gauss-Seidel (LU) NAS Parallel Benchmark [23] and wavefront-parallel algorithms. We utilize the Synch_p2p kernel, provided in the Intel Parallel Research Kernels (PRK) [16] to investigate the performance impact on this class of algorithms. The PRK suite consists of a set of common low level operations, and it has recently been released as open source [16]. PRK provides serial, OpenMP, and MPI implementations; for the purpose of this study we ported the MPI version of the Synch_p2p kernel to the OpenSHMEM programming model.

Synch_p2p implements a one-dimensional software pipeline. A two-dimensional array A of size $n \times m$ is distributed in vertical strips among the PEs. The matrix elements are updated through the stencil operation, $A(i,j) = A(i-1,j) + A(i,j-1) - A(i-1,j-1)$. This operation carries dependences in each of the spatial dimensions and is, therefore, not parallelizable in a straightforward manner. Parallelism is achieved by setting up pipelined execution. The first PE computes one partial

```
/* Let vector be an array that holds the grid values. */
/* We define the ARRAY macro to simplify indexing with halo elements. */
#define ARRAY(i, j) vector[i+1 + (j)*(segment_size+1)]

for (j = 1; j < n; j++) {
    /* I am not at the left boundary; wait for my left neighbor to send data */
    if (PE > 0) {
        shmem_ct_wait(ct, j);
        ARRAY(start[PE]-1, j) = dst[j];
    }

    for (i = start[PE]; i <= end[PE]; i++) {
        ARRAY(i, j) = ARRAY(i-1, j) + ARRAY(i, j-1) - ARRAY(i-1, j-1);
    }

    /* I am not on the right boundary; send data to my right neighbor */
    if (PE != NPES-1) {
        src[j] = ARRAY(end[PE], j);
        shmem_putmem_ct(ct, &dst[j], &src[j], 1 * sizeof(double), PE+1);
}
```

Listing 1.6. Pipelined parallel stencil kernel, using counting puts for point-to-point synchronization

row (fixed j) of updated elements. It then synchronizes with its right neighbor and proceeds to the second row. The neighboring process can now start with the update of its segment of the first row. Once the pipeline is filled, all PEs will be working in parallel. A code listing of the kernel using counting puts is shown in Listing 1.6.

In Figure 5, we show results from a strong scaling experiment, comparing the counting puts implementation with an implementation that uses explicit flags. For this experiment, we use a fixed matrix size of 12800×1280 and utilize 4 PEs per node to reduce noise generate by per-PE communication helper threads created by the Portals-on-InfiniBand runtime system. Threads are also pinned to cores to further reduce system noise. Results are reported in terms of the giga-FLOPs (floating point operations) per second achieved by the benchmark.

The total number of synchronizations required for each iteration of the Sync_p2p kernel increases with an increasing number of PEs while the computational work between synchronization points decreases. Because of this, synchronization cost is a significant factor in performance. From the results in Figure 5, we can see that the cost of synchronization when explicit flags are used is high, resulting in poor scaling. Counting puts eliminate the overhead of synchronization, significantly improving the parallel efficiency.

3.4 Impact of Problem Size on Performance Improvement

We now consider a fixed number of PEs and report the performance when the problem size is varied. To vary the problem size, we fix the length of the m dimension and vary the length of the n dimension. Figure 6 compares the performance for each problem size when counting puts and explicit flags are used. Experiments were run on 48 PEs, with 4 PEs per node. We note that the performance difference between the two implementations decreases with an increase in problem size. This is expected, as the number of synchronizations required per iteration depends only on the length of the second dimension. It is therefore the same for all sizes under consideration. The computational workload,

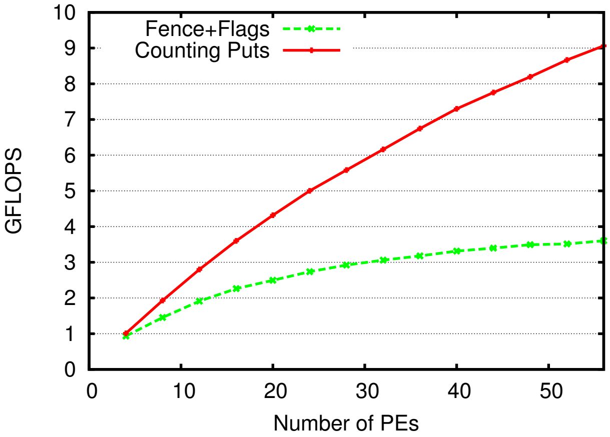

Fig. 5. Synch_p2p performance in GFLOPs/sec, for a strong scaling experiment with problem size of 12800 × 1280 and 4 PEs per node

however, increases, reducing the impact of synchronization cost. We note that the granularity of the algorithm could be increased by grouping rows together.

4 Related Work

Unified Parallel C (UPC) [22] is another PGAS parallel programming model, that provides capabilities similar to SHMEM. The current UPC language provides similar synchronization routines as SHMEM, with the addition of split-phase barriers and locks. A proposal to extend UPC with semaphores has been presented [9]. Semaphores would add a similar signaling capability to UPC put operations, and the authors demonstrated significant performance improvements across several platforms. An implementation of this extension is provided with Berkeley UPC [7]. UPC semaphores are implemented using carefully optimized active message and one-sided operations. The implementation approach we have presented utilizes receiver-side communication events to further reduce synchronization overheads.

Split-C [12] also provided signaling store operation through the :- assignment operation. A process that is the target of a signaling store operation can wait for a programmer defined number of bytes to arrive, but cannot distinguish among different update operations. Both the SHMEM counting puts and UPC semaphore extension allow this distinction by providing distinct synchronization objects. The Tera MTA [3], Cray XMT [14], and the Chapel programming language [11] also provide signaling store operations through full/empty bits that are associated with each word in memory, in the case of the MTA and XMT architectures, and distinct objects in the case of Chapel.

The ARMCI [18] one-sided communication library also provides a put-with-flag operation, that bundles a flag variable update with data movement.

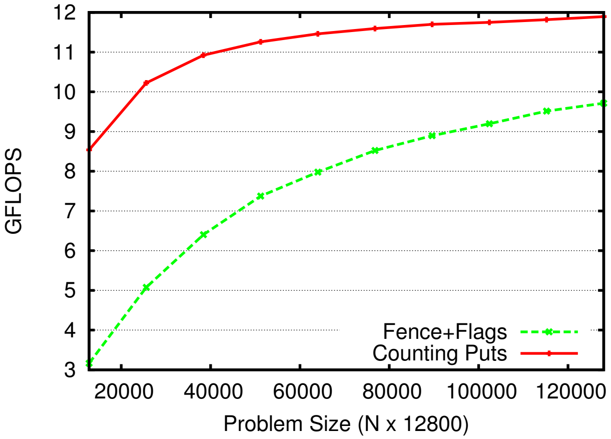

Fig. 6. Synch_p2p performance for various problem sizes with increasing first dimension N on 48 PEs, 4 PEs per node

Similarly, the GASPI [15] PGAS library provides a write-and-notify operation, that bundles an event notification with data movement. In both cases, the notification is performed by a write, rather than an atomic update. Thus, for algorithms that require many synchronizations, many flag and event variables would be needed. Depending on the number of variables needed, checking for completion can become costly and has the potential to negatively impact application data residency in the processor cache.

The Message Passing Interface (MPI) [17] provides both one-sided and two-sided messaging. Two-sided messaging effectively couples synchronization and data movement, since both sender and receiver participate in the communication operation. By bundling these two operations, two-sided messaging does not require additional operations for synchronization. However, in comparison with one-sided messaging, two-sided messaging incurs additional protocol overheads from the matching of send and receive operations. In addition, if the sender performs its send operation before the receiver performs its receive operation, the MPI library must buffer the unexpected message or delay data transmission. In contrast, PGAS programming models do not require message matching, buffering, or rendezvous protocols because remotely accessible memory in the global address space is necessarily posted before communication can occur and the sending process determines all communication parameters.

Active messages [8,13] provide a general-purpose mechanism for asynchronous operations that access memory at the target process. One-sided communication, can be implemented using active messages [8], and such an implementation can also notify the target process that the operation has been performed. Hardware support for RDMA has made it possible to implement one-sided operations directly in hardware without a target-side software agent. In this paper, we observe that hardware support for communication completion events also makes it possible to implement bundled one-sided communication and notification in hardware efficiently.

4.1 Concluding Discussion

We have presented a counting puts extension to OpenSHMEM, and discussed efficient implementations on a variety of networks, focusing on an implementation on top of the Portals network API. Experimental results indicate that the counting puts extension maps to an efficient implementation in Portals, and that it can offer a significant reduction in the overhead associated with point-to-point synchronization in SHMEM.

The proposed synchronization extension addresses an important need for SHMEM users, and it should be considered for inclusion in the OpenSHMEM standard. However, other synchronization mechanisms should also be considered, to provide a more flexible and efficient interface to users. Overlapping communication and computation is an important performance optimization, that can be used to hide communication costs. Counting puts can enable the user to achieve this overlap by periodically polling for data arrival. For some algorithms, global synchronization is needed. Non-blocking global synchronization is an increasingly popular primitive that is provided by several popular parallel programming models, including UPC [22] and MPI 3.0 [17]. The addition of a non-blocking, or split-phase barrier primitive could also help to address the synchronization needs of such applications.

Acknowledgements. We thank Tim Mattson and Rob van der Wijngaart of Intel Coporation, who developed the Parallel Research Kernels benchmark suite, and assisted us in porting the Sync_p2p benchmark to SHMEM.

References

1. OpenSHMEM implementation using portals 4. Website, `http://code.google.com/p/portals-shmem/`
2. Portals 4 open source implementation for InfiniBand. Website, `http://code.google.com/p/portals4/`
3. Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., Smith, B.: The Tera computer system. In: Proc. ACM Intl. Conf. on Supercomputing, ICS (June 1990)
4. Bariuso, R., Knies, A.: SHMEM user's guide. Tech. Rep. SN-2516, Cray Research, Inc. (1994)
5. Barrett, B.W., Brightwell, R., Hemmert, K.S., Pedretti, K.T., Wheeler, K.B., Underwood, K.D.: Enhanced support for OpenSHMEM communication in Portals. In: Hot Interconnects, pp. 61–69. IEEE (2011)
6. Barrett, B.W., Brightwell, R., Hemmert, S., Pedretti, K., Wheeler, K., Underwood, K., Riesen, R., Maccabe, A.B., Hudson, T.: The portals 4.0.1 network programming interface. Tech. Rep. SAND2013-3181, Sandia National Laboratories (April 2013)

7. Berkeley UPC: Berkeley UPC user's guide version 2.16.0. Tech. rep., U.C. Berkeley and LBNL (2013)
8. Bonachea, D.: GASNet specification, v1.1. Tech. Rep. UCB/CSD-02-1207, U.C. Berkeley (2002)
9. Bonachea, D., Nishtala, R., Hargrove, P., Yelick, K.: Efficient point-to-point synchronization in UPC. In: 2nd Conf. on Partitioned Global Address Space Programming Models, PGAS 2006 (October 2006)
10. Bruck, J., Ho, C.T., Upfal, E., Kipnis, S., Weathersby, D.: Efficient algorithms for all-to-all communications in multiport message-passing systems. IEEE Trans. Parallel Distrib. Syst. 8(11), 1143–1156 (1997)
11. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Intl. J. High Performance Computing Applications (IJHPCA) 21(3), 291–312 (2007)
12. Culler, D., Dusseau, A., Goldstein, S., Krishnamurthy, A., Lumetta, S., von Eicken, T., Yelick, K.: Parallel programming in Split-C. In: Proc., Supercomputing 1993, pp. 262–273 (1993)
13. von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active messages: a mechanism for integrated communication and computation. In: Proc. 19th Intl. Symp. on Computer Architecture, ISCA 1992, pp. 256–266 (1992)
14. Feo, J., Harper, D., Kahan, S., Konecny, P.: ELDORADO. In: Proc. 2nd Conf. on Computing Frontiers, CF 2005 (2005)
15. GASPI Consortium: GASPI: Global address space programming interface specification of a PGAS API for communication. Version 1.00 (June 2013)
16. Mattson, T., van der Wijngaart, R.: Parallel research kernels. Website (2013), `https://github.com/ParRes/Kernels`
17. MPI Forum: MPI: A message-passing interface standard version 3.0. Tech. rep., University of Tennessee, Knoxville (September 2012)
18. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In: Rolim, J., et al. (eds.) IPPS-WS 1999 and SPDP-WS 1999. LNCS, vol. 1586, pp. 533–546. Springer, Heidelberg (1999)
19. OpenSHMEM Consortium: OpenSHMEM application programming interface, version 1.0 (January 2012)
20. Reed, D., Kanodia, R.: Synchronization with event counts and sequences. Communications of the ACM 22(2), 115–123 (1979)
21. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. International Journal of High Performance Computing Applications (IJHPCA) 19(1), 49–66 (2005)
22. UPC Consortium: UPC language specifications, v1.2. Tech. Rep. LBNL-59208, Lawrence Berkeley National Lab (2005)
23. Yarrow, M., van der Wijngaart, R.: Communication improvement for the LU NAS parallel benchmark: A model for efficient parallel relaxation schemes. Tech. Rep. NAS-97-032, NASA Ames Research Center (1997)