One-Sided Append: A New Communication Paradigm For PGAS Models

James Dinan and Mario Flajslik
Intel Corporation
{james.dinan, mario.flajslik}@intel.com

ABSTRACT

One-sided append represents a new class of one-sided operations that can be used to aggregate messages from multiple communication sources into a single destination buffer. This new communication paradigm is analyzed in terms of its impact on the OpenSHMEM parallel programming model and applications. Implementation considerations are discussed and an accelerated implementation using the Portals 4 networking API is presented. Initial experimental results with the NAS integer sort benchmark indicate that this new operation can significantly improve the communication performance of such applications.

1. INTRODUCTION

Partitioned Global Address Space (PGAS) models – such as OpenSHMEM [1] and Unified Parallel C (UPC) [2] – are characterized by global data that is accessed through one-sided get, put, atomic update, and atomic read-and-update operations. This set of operations can produce efficient communication patterns because these operations require no coordination between sending and receiving processes. However, when coordination is required between processes, a strictly one-sided interface can be less efficient, as coordination must be constructed by the application using a series of one-sided operations to update coordination structures (e.g. locks, flags, or counters) in the global address space [3].

Parallel sorting algorithms, such as the parallel bucket sort algorithm used by the NAS Parallel Benchmarking (NPB) Integer Sort (IS) benchmark [4], perform periodic many-to-many communications to exchange items during the sorting process. The number of items communicated between peers varies throughout the sorting process and is dependant on the input data. When such algorithms are implemented using PGAS models [5], users must construct a method for pre-posting adequate buffer space to allow remote processes to write data through one-sided put operations. A global coordination step can be used to collectively establish communication buffers, so that a fixed amount of buffer space can be posted for each sender-receiver pair. Alternatively, a large, shared buffer can be posted and processes can append or push items into available space at the tail of the buffer.

```
start_disp = shmem_fadd(push_counter, src_size,
                        dest_pe);
shmem_putmem(((char *) dest_buf) + start_disp,
             src_buf, src_size, dest_pe);
```

Listing 1: One-sided append implementation of parallel sorting key exchange step.

Such one-sided append operations can be implemented on top of OpenSHMEM as shown in Listing 1. Where a shared "push" counter is updated using an atomic fetch-and-add (FADD) operation to reserve space at the tail of a shared buffer and data is then written to this shared buffer using a one-sided put operation.

In this approach, a globally accessible counter is used to coordinate among multiple processes that write data to adjacent locations in a shared buffer. This approach has the advantage of using a shared buffer, which can use memory efficiently and avoid global communication. However, the coordination step requires an additional round-trip fetch-and-add communication operation, doubling the number of communication operations performed.

We propose a new PGAS communication operation – one-sided append – which bundles coordination and communication enabling PGAS runtime systems to optimize this pair of actions. In Section 2 we discuss an API extension to provide one-sided append in the OpenSHMEM library. We discuss implementation considerations in Section 3, and describe an accelerated implementation using the Portals 4 network programming interface [6]. We demonstrate the application of one-sided append to the NAS integer sort benchmark and present an early evaluation of our prototype Portals 4 implementation in Section 4. Our results show that one-sided append can provide up to a 500% improvement in performance when key exchange messages are small.

2. ONE-SIDED APPEND EXTENSION TO OPENSHMEM

We define a new SHMEM object, `shmem_oct_t`, which establishes the offset counter (OCT) needed to support one-sided append operations. In Listing 2, we propose an application programming interface (API) extension to OpenSHMEM that utilizes an OCT to support one-sided append.

OCTs are created and destroyed collectively. When an OCT is created, the user supplies a pointer to the symmetric buffer that will be appended to. All SHMEM processes, or PEs, in a given collective call to `shmem_oct_create()` must provide local point-

```
/* Offset counter (OCT) creation/destruction functions (collective) */
void shmem_oct_create(shmem_oct_t *oct, void *buffer, size_t len);
void shmem_oct_destroy(shmem_oct_t *oct);

/* Appending put, a.k.a "push", communication operation (one-sided) */
void shmem_push(shmem_oct_t oct, const void *src_buffer, size_t len, int pe);

/* Offset counter update/query routines (local) */
void shmem_oct_reset(shmem_oct_t oct);
size_t shmem_oct_get(shmem_oct_t oct);
```

Listing 2: OpenSHMEM one-sided append API.

ers to the same symmetric buffer that was allocated statically in the static data segment or dynamically, e.g. using `shmalloc()`. This requirement ensures that the runtime system can store a single symmetric pointer for use in append operations, avoiding an OCT-related structure whose size is proportional to the number of PEs. The create function could also be defined to allocate a symmetric buffer that will be used for append operations; however, performing allocation outside of the OCT interface provides greater flexibility. For example, the proposed interface can be used with a symmetric buffer located in the static data segment.

A one-sided append is performed by a call to either the blocking `shmem_push()` function or its nonblocking counterpart. A push operation appends the source buffer data to the append buffer associated with the given OCT at the destination PE. Data is written contiguously, starting at the first free memory location in the buffer. While push operations do not need to be aligned, it is likely that appending messages whose size results in aligned accesses will provide higher performance.

2.1 Offset Counter Semantics

Several options are available for handling the situation where the space available at the destination PE is not sufficient to hold the pushed data. A simple approach is to leave this behavior undefined or to define it as an erroneous program. Alternatively, push can be defined to truncate the data, storing only the portion that fits in the remaining space of the destination PE's buffer. This semantic may make the interface easier to use and more productive for application developers. In order to support a truncation model, the push operation must be extended to return the number of bytes that were pushed and the creation routine must also require that the buffer length argument be identical at all PEs for a given OCT. This latter requirement ensures that PEs do not need to store an array of buffer lengths for each PE to determine when truncation occurs.

The OCT can be reset to the beginning of the append buffer through a call to `shmem_oct_reset()`. Resetting the OCT is a local operation, and users must ensure that the reset operation is correctly ordered with respect to push operations performed by remote PEs on the given OCT. The offset of the next free location in the local append buffer can be queried through a call to `shmem_oct_get()`. This operation does not guarantee that pushed data is available to read; users must synchronize these accesses through other means, e.g. a call to `shmem_barrier()`. Alternatively, a counting push operation could be defined using the approach detailed in [3], which provides a separate counter to track completed operations. If bytes completed (rather than messages completed) are tracked, the user can check for equality between of the offset counter and the completion counter. When both counters are equal, all data in the append buffer up to the current offset counter is valid and can be read.

```
typedef struct {
  ptl_pt_index_t   pt;
  ptl_handle_me_t  me;
  ptl_match_bits_t match;
  ptl_handle_ct_t  ct;
} shmem_oct_t;
```

Listing 3: Offset counter (OCT) structure used in the Portals 4 SHMEM implementation.

3. IMPLEMENTATION DISCUSSION

The SHMEM push operation can be implemented on top of the existing OpenSHMEM API using the approach shown in Listing 1. In this approach, a symmetric offset counter is allocated in the call to `shmem_oct_create()` and is atomically updated using fetch-and-add operations. This represents a general implementation technique and the SHMEM runtime can internally use low-level RDMA operations provided by a variety of HPC system interconnects.

While one-sided append can be implemented on top of most PGAS interfaces, providing a specialized API enables optimizations within the communication subsystem, as illustrated in Figure 1. We next explore an accelerated implementation that leverages capabilities defined by the Portals 4 network programming interface [6] to provide an efficient implementation. In comparison with an implementation on top of the OpenSHMEM API, where the sender must remotely manipulate the offset counter, accelerated implementations can utilize offload capabilities at the receiver to integrate offset counter manipulation with processing of the push message.

3.1 Receiver-Managed Implementation Using Portals 4

As shown in Figure 1, receiver-managed implementations, where the receiver manages the offset into the receive buffer, can significantly improve the efficiency of one-sided append, or push, operations. Such an implementation reduces the amount of communication messages by half, thus providing better latency and throughput. We use Portals' locally managed offset to achieve a receiver-managed implementation of one-sided append, and the internal structure used to store OCT metadata is shown in Listing 3. At the receiver, the sender-provided portal table (PT) index and match bits from the message header are used to look up a match list entry (ME) corresponding to an OCT. The ME is configured to use a locally managed offset, which instructs Portals to append incoming data to the ME's buffer. An additional Portals counter (CT) is used to expose the locally managed offset value for application-level queries. We explain each of these aspects of the implementation in greater detail below.

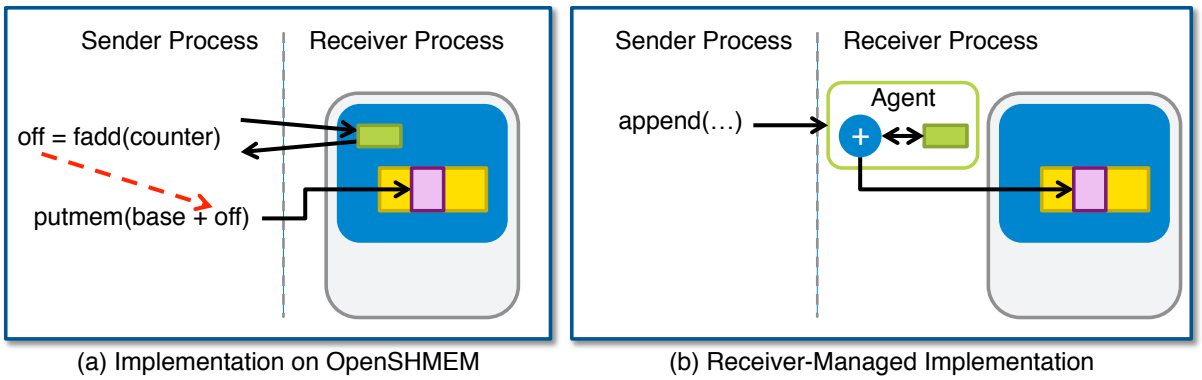

Figure 1: Sender (a) and receiver (b) managed implementations of one-sided push. Receiver-managed implementations access the offset counter locally reducing the number of communication operations performed by the sender.

OCT creation is collective across all PEs, much like the creation of SHMEM counters in [3]. Global coordination is necessary so that all instances of the OCTs on all PEs agree to use the same portal table match bits values. Tradeoffs can be made to balance the number of portal table entries assigned to supporting one-sided append operations, and the length of the match lists associated with these portal table entries, as message processing overheads can increase with the length of the match lists. Identical portal table and match bits allow the sender to address the remote OCT using its own instance of the corresponding OCT, eliminating the need to store separate location information for each PE. Similarly, freeing the OCT is also a global collective operation to guarantee correct reuse of portal table and match bits values.

During OCT creation, each PE also sets up a match list entry (ME) for its own append buffer. The match list entry is configured to address only the buffer space passed into the `shmem_oct_create()` function, and `PTL_ME_MANAGE_LOCAL` option is set up on the match list entry to enable the locally managed offset feature. Additionally, match bits are set up to bind the match list entry to the correct buffer, and a portals counter is attached to the ME to record counting events. The portal table and match bits pair must be unique for each OCT. This can be achieved by keeping the match bits unique through a counter that is incremented in `shmem_oct_create()`, every time that function is called.

Depending on the desired semantic, SHMEM applications might be expected to provide enough buffer space in the initial call to `shmem_oct_create()` that covers all data that might be pushed into the buffer. For the remainder of this paper, we assume a model where behavior is undefined in case of buffer overflow. In our implementation, Portals is configured to truncate data that does not fit in the remaining buffer space (which can be zero), and the sender is not made aware of this failure to avoid retransmitting the messages repeatedly. In this model, it is up to the SHMEM application to detect and recover from a buffer overflow error.

The API proposed in Listing 2 also provides functions to reset the OCT object and query the offset counter inside the OCT object. Portals 4 API does not provide a way to access the offset counter in MEs with locally managed offset. However, it is possible to attach a Portals counter to the ME, and set it to count the number of bytes received into the append buffer by setting the `PTL_ME_EVENT_CT_BYTES` ME option. We keep this Portals counter inside the OCT object (see Listing 3) as a proxy for the offset counter. While the two counters do match eventually, the actual offset counter is incremented at the beginning of push message processing, and the ME byte counter is updated once all of the data has been received. However, this is okay because applications must use some form of synchronization before reading the counter, and after synchronization the ME byte counter and the actual offset counter have the same value.

The synchronization before reading the counter is required because of lack of ordering guarantees. There is no implied guarantee that the data bytes received are stored in the append buffer in order. For example, in a series of three push operations, the second can arrive after the others. The application must use some other means of synchronization (e.g. a barrier) to guarantee that all of expected data has been placed in the append buffer.

3.2 Discussion of Implementation Alternatives

In Portals 4, the locally managed offset counter is not exposed through the portals interface. An extension to the Portals API that allows access to this counter would enable a more straightforward implementation of `shmem_oct_get()` function. It would also allow an implementation that waits until the number of allocated bytes matches the number of received bytes. When the two counters match, the receiver is guaranteed that all those bytes are placed in the buffer in order, which sometimes may remove a requirement for extra synchronization.

With direct access to the offset counter, we could configure the ME counter to count the number of received messages (instead of bytes). Sometimes, applications know to expect each PE to append exactly one message, or to expect a fixed number of messages. The ME Portals counter could be used to wait for a specified number of messages, thus potentially removing a requirement for a barrier.

In our implementation, push buffers are not necessarily required to be symmetric across all PEs. The proposed SHMEM extension requires symmetric buffers to ensure portability; however, if the user passes a non-symmetric buffer to `shmem_oct_create()` our implementation will still behave correctly. This is possible because the buffers are addressed at the sender using the OCT object, rather than a symmetric address. Non-symmetric buffers would provide additional flexibility when managing and mapping memory, which is not available through an implementation append on top of OpenSHMEM. For example, one could be much more memory efficient if using a single PE as an aggregator for all data using push oper-

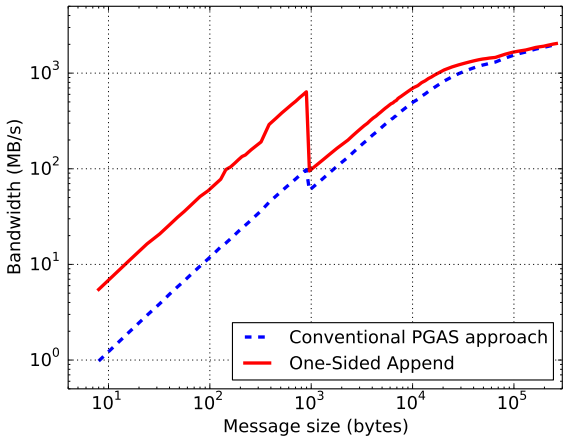

Figure 2: Bandwidth performance comparison between baseline and push implementations of key exchange on 16 PEs.

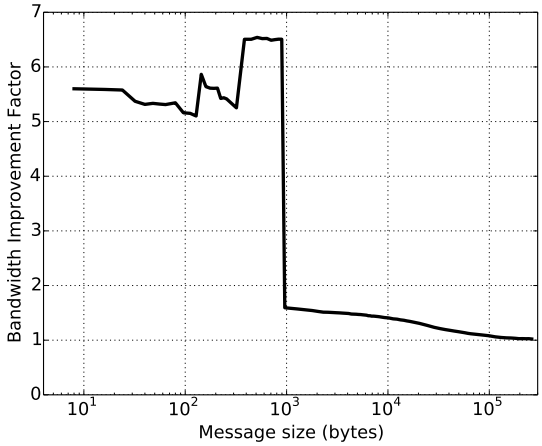

Figure 3: Bandwidth improvement factor from using one-sided push. Derived from Figure 2 by dividing the push bandwidth measurement by the baseline.

```
for (i = 0; i < num_pes; i++) {
  int k1 = send_offset[i];
  int k2; // target offset
  k2 = shmem_int_fadd(&counter,
                      snd_cnt[i], i);
  shmem_int_put(key_buf2+k2, key_buf1+k1,
                snd_cnt[i], i);
}
```

Listing 4: Baseline integer sort key exchange code snippet.

```
shmem_oct_create(&off, key_buf2, BUF_SIZE);
...
for (i = 0; i < num_pes; i++) {
  int k1 = send_offset[i];
  shmem_push(off, key_buf1+k1, snd_cnt[i], i);
}
...
shmem_oct_free(&off);
```

Listing 5: Integer sort key exchange using one-sided push.

ations. However, nonsymmetric allocations are not currently supported by the OpenSHMEM specification.

4. EVALUATION

Appending data to a remote buffer is performance critical for applications that perform parallel sorting. One such application is the integer sort (IS) benchmark in the NAS parallel benchmark suite [4], which performs parallel bucket sort. In such algorithms, processes must periodically exchange data items based on the sorting keys associated with those items.

The communication-intensive key exchange step of parallel bucket sort using OpenSHMEM is shown in Listing 4. In this algorithm, the sending PE first uses atomic fetch-and-add operation to increment a counter at the receiver and obtain an offset into the receiving buffer. The counter controlling the offset into the receiving buffer is located at the receiver, but it is managed by the senders through one-sided, atomic fetch-and-add operations. After obtaining the target buffer offset, the sender does a one-sided put operation to place the sorting keys into the receiving buffer at the correct offset. This algorithm requires two communication operations per receiver. Additionally, as shown in Figure 1, there is a data dependency between the fetch-and-add and the following put, forcing the fetch-and-add to complete before the put can be issued.

The same sort key exchange algorithm can be implemented using one-sided push, as shown in Listing 5. Some time after allocating the receive buffer, the PE sets up a symmetric offset object for that buffer enabling the one-sided push to be used with the receive buffer. In the key exchange loop, the sender makes a single call to `shmem_push()` for each receiver. The offset into the receive buffer is managed at the receiver side, and there are no more data dependencies at the sender. Lack of data dependencies allows for significant performance gains in the small message case. For small messages that can be buffered in the network, `shmem_push()` returns immediately, allowing the loop to proceed to the next iteration.

Figure 2 shows bandwidth performance of the one-sided push key exchange algorithm in Listing 5 and the baseline algorithm in Listing 4. The relative bandwidth improvement factor is shown in Figure 3. These experiments were performed on key exchange algorithms shown in Listing 4 and Listing 5, distributed over 16 PEs. The results were generated on a 16 node InfiniBand* cluster using a SHMEM over Portals 4 [7] in combination with the Portals 4 on InfiniBand reference implementation [8]. For small messages that can be buffered in the network, one-sided push implementation provides more than 5x improvement. Small messages (less than 1kB) are buffered in the Portals layer, and both `shmem_int_put()` and `shmem_push()` return immediately for those small messages. Larger messages require Portals layer to complete the operation remotely before it can release the local buffer and return from a blocking operation. Bandwidth improvement for a 1kB size message is about 60%, and it tapers off for very large messages.

5. CONCLUSION

One-sided append is a new extension to PGAS models that aggregates contributions from multiple PEs into a single buffer at the receiving PE. We have presented an API definition and initial implementation discussion of one-sided append. Through the Portals

4 network programming interface, we showed that the new interface provides opportunities for hardware acceleration that are not available when append operations are implemented on top of an existing one-sided communication interface. Using SHMEM push, an instantiation of one-sided append in the OpenSHMEM API, to accelerate the key exchange step in the NAS integer sort parallel benchmark, we observed speedups in excess of 500% for messages below 1kB.

6. REFERENCES

[1] *OpenSHMEM Application Programming Interface, Version 1.1*, Online: http://openshmem.org, Jun. 2014.

[2] UPC Consortium, "UPC language and library specifications, version 1.3," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-6623E, 2013.

[3] J. Dinan, C. Cole, G. Jost, S. Smith, K. D. Underwood, and R. W. Wisniewski, "Reducing synchronization overhead through bundled communication," in *OpenSHMEM*, ser. Lecture Notes in Computer Science, S. W. Poole, O. Hernandez, and P. Shamis, Eds., vol. 8356. Springer, 2014, pp. 163–177.

[4] 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*, vol. 5, no. 3, pp. 63–73, 1991.

[5] "NAS parallel benchmarks for OpenSHMEM, version 1.0a," Online: http://openshmem.org/site/Downloads/Examples, Aug. 2014.

[6] B. W. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson, "The portals 4.0.1 network programming interface," Sandia National Laboratories, Tech. Rep. SAND2013-3181, April 2013.

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

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