

# Evaluating OpenSHMEM Explicit Remote Memory Access Operations and Merged Requests

Swen Boehm<sup>(✉)</sup>, Swaroop Pophale, and Manjunath Gorentla Venkata

Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA  
{bohms,pophale,manjugv}@ornl.gov

**Abstract.** The OpenSHMEM Library Specification has evolved considerably since version 1.0. Recently, non-blocking implicit Remote Memory Access (RMA) operations were introduced in OpenSHMEM 1.3. These provide a way to achieve better overlap between communication and computation. However, the implicit non-blocking operations do not provide a separate handle to track and complete the individual RMA operations. They are guaranteed to be completed after either a `shmem_quiet()`, `shmem_barrier()` or a `shmem_barrier.all()` is called. These are global completion and synchronization operations. Though this semantic is expected to achieve a higher message rate for the applications, the drawback is that it does not allow fine-grained control over the completion of RMA operations.

In this paper, first, we introduce non-blocking RMA operations with requests, where each operation has an explicit request to track and complete the operation. Second, we introduce interfaces to merge multiple requests into a single request handle. The merged request tracks multiple user-selected RMA operations, which provides the flexibility of tracking related communication operations with one request handle. Lastly, we explore the implications in terms of performance, productivity, usability and the possibility of defining different patterns of communication via merging of requests. Our experimental results show that a well designed and implemented OpenSHMEM stack can hide the overhead of allocating and managing the requests. The latency of RMA operations with requests is similar to blocking and implicit non-blocking RMA operations. We test our implementation with the Scalable Synthetic Compact Applications (SSCA #1) benchmark and observe that using RMA operations with requests and merging of these requests outperform the implementation using blocking RMA operations and implicit non-blocking operations by 49% and 74% respectively.

---

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<http://energy.gov/downloads/doe-public-access-plan>).

## 1 Introduction

OpenSHMEM 1.3 [1] introduced implicit non-blocking *puts* and *gets* to the existing library *Application Programming Interface* (API). The semantics of these operations allow to post the operation, and later wait for its completion. This has advantages over previous blocking semantics as overlap between the communication and other operations can be achieved. These operations are considered complete only after a remote memory update is guaranteed through the use of a *shmем\_quiet*, *shmем\_barrier* or a *shmем\_barrier\_all*. Since *shmем\_quiet*, *shmем\_barrier*, and *shmем\_barrier\_all* are global completion operations i.e., *shmем\_quiet* completes all outstanding memory update operations by a particular Processing Element (PE) and *shmем\_barrier* completes all outstanding memory update operations on all PEs (and synchronizes them), it can have a significant performance impact on applications that only require finer grained completion.

This paper proposes the introduction of non-blocking data transfer calls with explicit requests, the ability to use single request for multiple operations, and interfaces to merge multiple requests. Explicit requests provide the capability of tracking individual data transfer operations. The option to group related RMA operations together into a single request handle provides flexibility to the programmer and can improve the application performance.

As we often see in scientific code, a series of updates are made during the computation phase and are written during the communication phase. Most updates need to happen together to enable the next set of computations. Such updates can be merged together to enable easy checking for the user. This has many performance as well as productivity implications. This approach may greatly simplify how users write their code, replacing multiple request handles by a single request handle. The performance advantage comes from the fact that testing completion of a single handle is much more cost efficient than either checking individual handles or executing mass memory updates via *quiet* or *barrier* that will only return after **all** pending local and remote memory updates are processed.

In Sect. 2, we first motivate the scenarios where these interfaces are useful. In Sect. 3, we provide details of the interfaces introduced. In Sect. 4, we discuss the details of our implementation. In Sect. 5 we modify the application kernels and benchmarks to demonstrate the usability, productivity, and performance advantages of the interfaces. We discuss the results of Sect. 5 in depth in Sect. 6. Related work in this context is covered in Sect. 7 and our next steps are discussed in Sect. 8.

## 2 Motivation

The traditional OpenSHMEM programming model is based on the foundation of maximum computation-communication overlap through fast one-sided RMA operations that do not require the involvement of the destination PE. Implicit

non-blocking calls were introduced in OpenSHMEM 1.3 [1]. These calls provide many advantages of non-blocking calls except the ability to track their completion. Even for 1.3 semantics, completion is guaranteed by either using a *quiet* or a *barrier*. Explicit non-blocking calls overcome this pitfall by providing fine grained control through individual request handles. The following are the scenarios where having merged handles for explicit RMA operations can be both advantageous and performant.

## 2.1 Use Case 1: OpenSHMEM Threads

As the OpenSHMEM Specification evolves to incorporate thread safety and a threading model, it becomes critical to define a synchronization mechanism within threads of a single PE. Many operations distributed within the threads may require sequential consistency. Merging handles for communication by a single thread allows for easy ordering of operations when compared to managing individual communication calls with explicit handles.

The Cray Threads proposal [4] offers a thread safe threading model that requires registering of threads after initializing threading support via *shmem\_thread\_register* at the start of a multithreaded OpenSHMEM program. Similarly, a *shmem\_thread\_unregister* is required to be called by threads that have registered via the *shmem\_thread\_register* call when threading support is no longer required within the OpenSHMEM program. Registering a thread that may make OpenSHMEM library calls during the lifetime of the program provides a means to track communication originating from that thread. This threading model also defines a *shmem\_thread\_quiet* as a means to coordinate activities between different threads of a single PE. Through our approach, a single handle can represent a collection of RMA operations made by a thread, thus allowing concurrency between non-related RMA operations issued by the same or different thread belonging to the same PE. We also eliminate the need for introducing, implementing, and maintaining three library calls which is an added bonus.

Dinan et al. [5] introduce *contexts* as a way to eliminate interference between threads by generating independent streams of communication operations that enable the OpenSHMEM library to map operations generated by threads to private communication resource sets. *Contexts* are intended to provide thread isolation and a greater control over ordering of operations. This can improve communication and computation overlap. The very same effect can be achieved with greater overlap opportunity by introducing non-blocking explicit RMA operations and providing the facility of merging related updates to a single request. The advantage of our approach is that many of the concepts already exist in other programming models and libraries, thus leading to better acceptance and use by the OpenSHMEM user community.

## 2.2 Use Case 2: Defining Patterns

The merging of the requests are particularly useful for communication and computation patterns where it is required to track a group of operations, and also

require opportunity to overlap the operations with computation. For example in a stencil computation operation, each phase of communication within a *sweep* can be merged into a single request. Since the results are not required till the next time-step, the communication can progress asynchronously with other communication or computation operations. Also, the user does not need to test for the completion of all the individual communications.

### 2.3 Use Case 3: Defining New Collectives

The RMA operations with explicit requests along with merging can provide a way to define customized one-sided collectives with ability to asynchronously progress the collectives. For example, currently *broadcast* in OpenSHMEM is restricted to updates from a *root* PE to other PEs defined by a regular (power-of-two stride) *active-set*. If a program frequently needs to update an irregular set of PEs, this might be encapsulated in a single merged-handle. Following the same logic, other customized non-blocking collectives are also possible. Moreover, this approach provides a means of providing overlap between collectives that are not using/updating the same *symmetric objects*.

## 3 API and Semantics for RMA Operations with Requests

In this section, first, we introduce the interfaces required for non-blocking RMA operations with requests. We then look at two possible ways to merge the requests. One way is to create a single merged request handle (which is a collection of requests), and the other approach is to merge existing requests into a single request.

### 3.1 Explicit Non-blocking RMA Operations

The interfaces for the *Put* operations are in Box 1, and the *Get* operations are in Box 2. They are used for transferring data from the origin PE to the target PE (*pe*) and from the destination PE to the origin PE respectively. The source of the data is passed as the *source* and the target buffer is passed as the *target*. The handle to track the *Put* operation is created by the library and returned with *handle*.

```
shmem_TYPE_put_nbe (TYPE *target, const TYPE *source, size_t nelems, int pe,
shmem_request_handle_t **handle);
shmem_putSIZE_nbe (TYPE *target, const TYPE *source, size_t nelems, int pe,
shmem_request_handle_t **handle);
```

**Box 1.** *Put* operations with requests

These operations return after initiating the *Put* (or *Get*) operation, but not necessarily before copying data out of the source variable/array. These semantics are similar to implicit RMA operations introduced in OpenSHMEM 1.3 [1].

```
shmem_TYPE_get_nbe (TYPE *target, const TYPE *source, size_t nelems, int pe,
shmem_request_handle_t **handle);
shmem_getSIZE_nbe (TYPE *target, const TYPE *source, size_t nelems, int pe,
shmem_request_handle_t **handle);
```

**Box 2.** *Get* operations with requests

However, the difference is in the completion of operations. The RMA operations with requests are required to call the *Wait* (Box 3) function to guarantee completion, or they can use the *Test* (Box 3) function to query the status of the operations.

```
void shmem_wait_req( shmem_request_handle_t *handle);
void shmem_test_req( shmem_request_handle_t *handle);
```

**Box 3.** Wait and Test operations for completing and testing the status of requests, respectively.

### 3.2 Merging RMA Request Handles

The RMA interfaces that take in requests that represent more than one operation is shown in Box 4. Using this interface, the user provides a hint to the library about usage of the data from the operations. It indicates that the user expects to group a set of RMA operations, which can be synchronized and completed simultaneously. The library can optimize by allocating independent network resources that can be independently synchronized and flushed for completion.

```
shmem_TYPE_RMA_nbe_multiple(TYPE *target, const TYPE *source, size_t
nelems, int pe, shmem_request_handle_t **handle);
```

**Box 4.** RMA Operation with Merged Request Handles

The interface for merging already existing requests is shown in Box 5. This interface is useful for tracking and completing already existing groups of RMA operations. The user has the flexibility to cherry-pick RMA requests that may be grouped together for maximum overlap. Similar to RMA operations with requests, these operations are completed using the *Wait* operation.

```
shmem_merge_requests(int num_req, shmem_request_handle_t **ReqArray,
shmem_request_handle_t **request );
```

**Box 5.** Interface for Merging the Requests

As merged handles are just a medium to provide a single request for multiple related RMA operations, they themselves do not impose any restrictions on the programmer when used alongside other OpenSHMEM API. All explicit non-blocking calls will complete after a *shmem\_quiet*, *shmem\_barrier*, or *shmem\_barrier\_all* is called but the user must call *shmem\_wait\_req* to release the request handle. This provides for a cleaner usage and better code readability as the user can easily match RMA operations to their corresponding *waits*. The use of *shmem\_fence* has no effect on the ordering of explicit non-blocking RMA.

## 4 Implementation Using UCX

The implementation of explicit and merged non-blocking RMA operations is done in the OpenSHMEM reference implementation. The reference implementation can use two different networking libraries. Figure 1 shows an overview of the dependencies. One is GASNet and the other is Unified Communication X (UCX) [14].

UCX is a middleware that provides a portable API that targets different underlying networking components. By providing a highly performant API framework, UCX exposes the constructs for implementing various programming models such as Message Passing Interface (MPI) and Partitioned Global Address Space (PGAS).

UCX is comprised of three major API frameworks. These frameworks can be used independently of each other. They include UC-Services (UCS) - provides services and common utilities, UC-Transports (UCT) - provides low level API for hardware transports, and UC-Protocols (UCP) - provides high level API that implement different protocols.

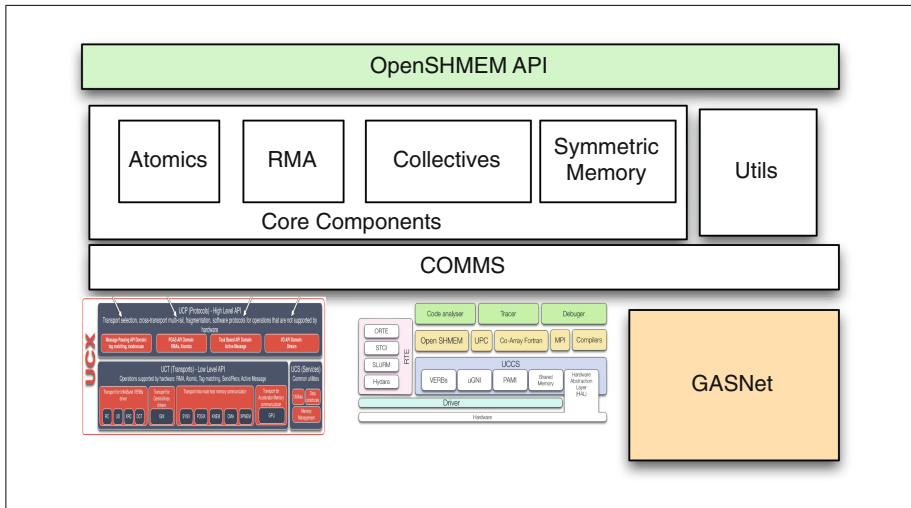


Fig. 1. Various components in the *OpenSHMEM* reference implementation

UCT abstracts the arduous details of the underlying hardware, thus providing a low-level API for implementing higher-level protocols. The API provides the necessary functionality for communication context management, device specific memory allocation and management, interfaces for various types of messages, remote memory access (RMA), *Atomic Memory Operations* (AMO), active messages, and collectives. The API is driven by the interconnect manufacturers.

UCP is layered over UCT and provides an abstraction of higher-level protocols. These can be used by programming models such as MPI and PGAS. UCP initializes the UCX library, allows for message fragmentation, and provides multi-rail communication.

To implement explicit RMA operations and merged requests, the UCX networking layer is used. While the OpenSHMEM reference implementation is setting up the symmetric heap and manages PEs, the RMA operations map directly to UCP functions. Therefore the explicit non-blocking operations are implemented in UCP with a small wrapper in OpenSHMEM.

## 5 Evaluation

### 5.1 Experimental TestBed

#### 5.1.1 System

We run our experiments on a 16 node SGI cluster with Mellanox ConnectX-4 VPI adapter card, EDR IB (100 Gb/s) and 100 GbE, a single-port QSFP, and PCIe3.0  $\times$  16. Each node comprises of two NUMA nodes with two sockets each and 10 cores per socket. Each of 40 CPUs is an Intel Xeon E5-2660 v3 operating at 2.6 GHz.

#### 5.1.2 Application Kernels and Benchmarks

For evaluation, we use micro-benchmarks and application kernels. The details of the application kernel is provided in Sect. 5.3. Here we present the experimental results and discuss them in detail in Sect. 6.

For evaluating the latency, bandwidth, and message rate, we modify benchmarks from OSU [15]. The modifications include changing the *shmem* interfaces to use non-blocking implicit and explicit RMA operations.

We modify the latency benchmark to mimic a ping-pong exchange. The ping-pong benchmark first sends the data from origin PE to remote PE. The remote PE waits for data using *shmem\_wait* on the last byte of the data, then sends a response to the origin PE. Though this approach may not reflect the arrival of the complete message for networks that do not guarantee in-order delivery, for Mellanox’s InfiniBand network with Reliable Connection (RC) transport protocol, in-order delivery is guaranteed.

## 5.2 Performance Evaluation of RMA Operations with Requests and Merged Requests Using Micro-Benchmarks

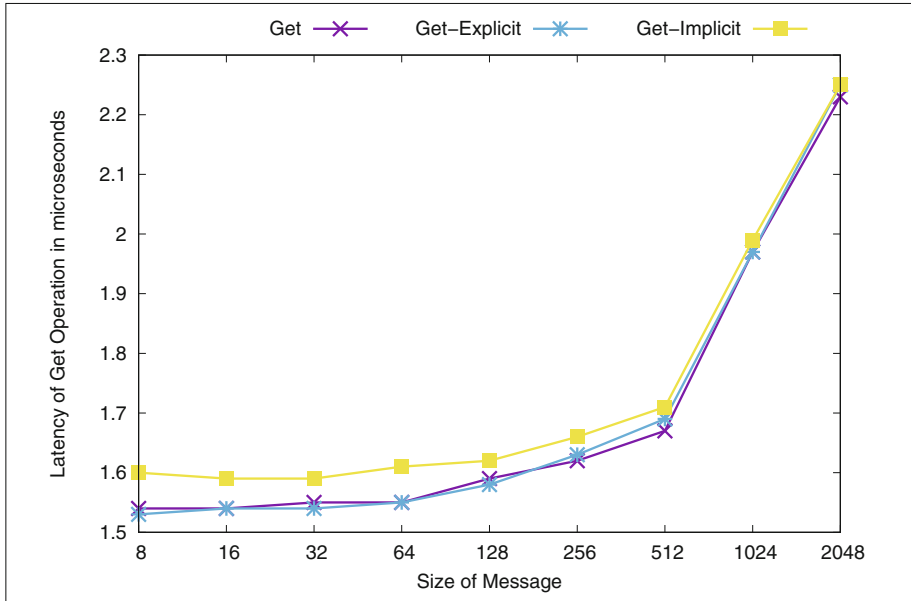
### 5.2.1 Latency of Get Operations

In this experiment, the performance of *shmem\_getmem*, *shmem\_getmem\_nbi*, and *shmem\_getmem\_nbe* operations is compared. The origin PE issues the *Get* operation, and waits for completion. In case of *shmem\_getmem*, the data is updated when the call returns. In the case of *shmem\_getmem\_nbi*, and *shmem\_getmem\_nbe*, it waits for *shmem\_quiet* and *shmem\_wait\_req* to complete respectively. Figure 2 shows that the latency of all *Get* operations are similar.

To understand the performance impact of global completion (*shmem\_quiet* and *shmem\_barrier*) used for completing implicit operations, we modify the *Get* benchmark to issue multiple *Get* operations. The origin PE issues *Get* operations to multiple PEs, and waits for completion only on one PE. From Fig. 3 we observe that the performance of RMA operations with requests outperform (as expected) both implicit non-blocking RMA operations and blocking RMA operations.

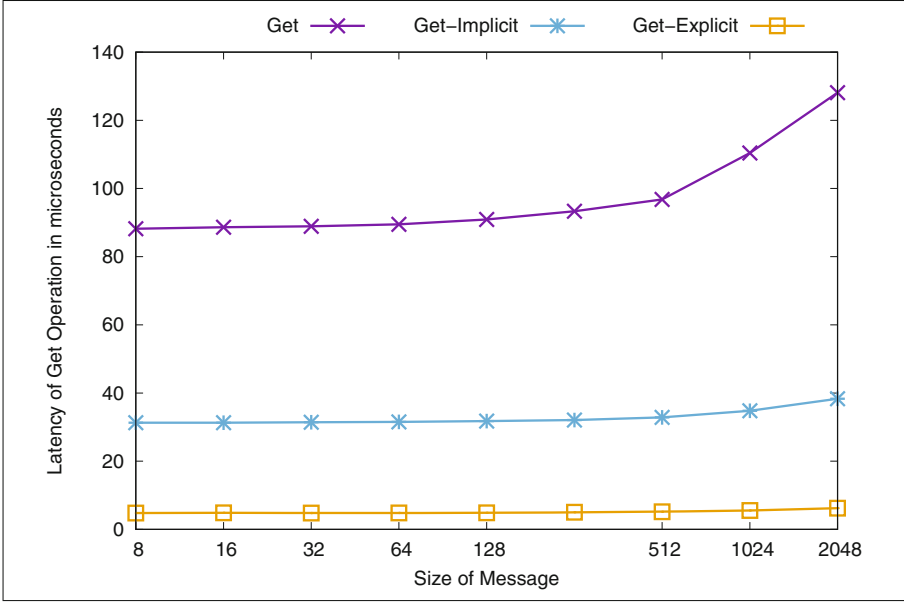
### 5.2.2 Ping-Pong Latency

Figures 4 and 5 compare the round trip time for *shmem\_put*, *shmem\_put\_nbi*, and *shmem\_put\_nbe* for small and large messages respectively. The origin PE sends a *ping* using *shmem\_put*, *shmem\_put\_nbi*, or *shmem\_put\_nbe*, and the destination

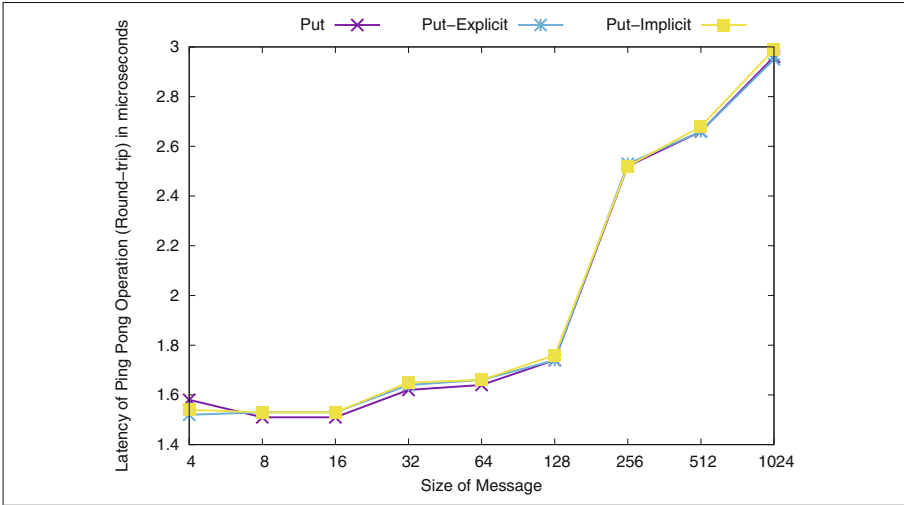


**Fig. 2.** Comparing performance of *shmem\_getmem*, *shmem\_getmem\_nbi*, and *shmem\_getmem\_nbe*



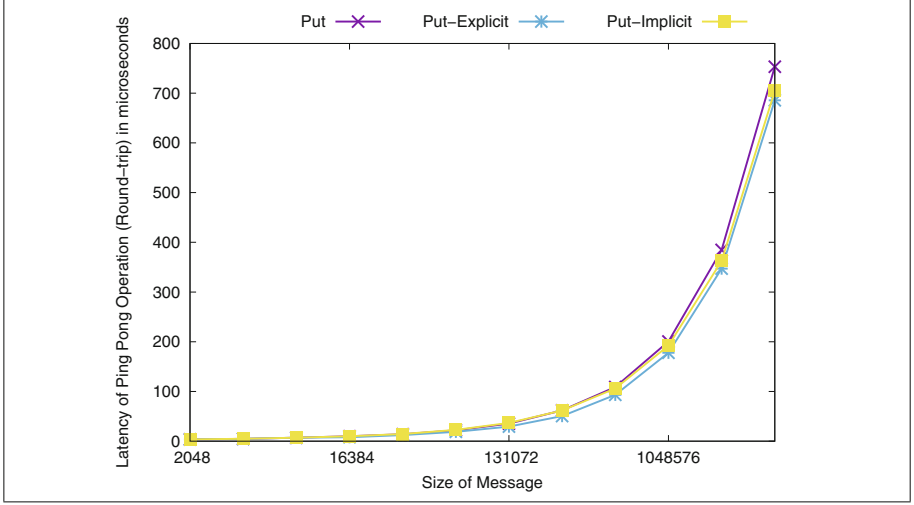


**Fig. 3.** Comparing performance of OpenSHMEM OSU *shmем* *get* many benchmark using 64 PEs



**Fig. 4.** Roundtrip latency using put-based ping-pong benchmark for small messages

PE and then waits on a corresponding *pong* using *shmем\_int\_wait\_until*. On receiving the *ping*, the destination PE responds with a *pong* through a *Put*. The target PE waits on the last byte of the message.



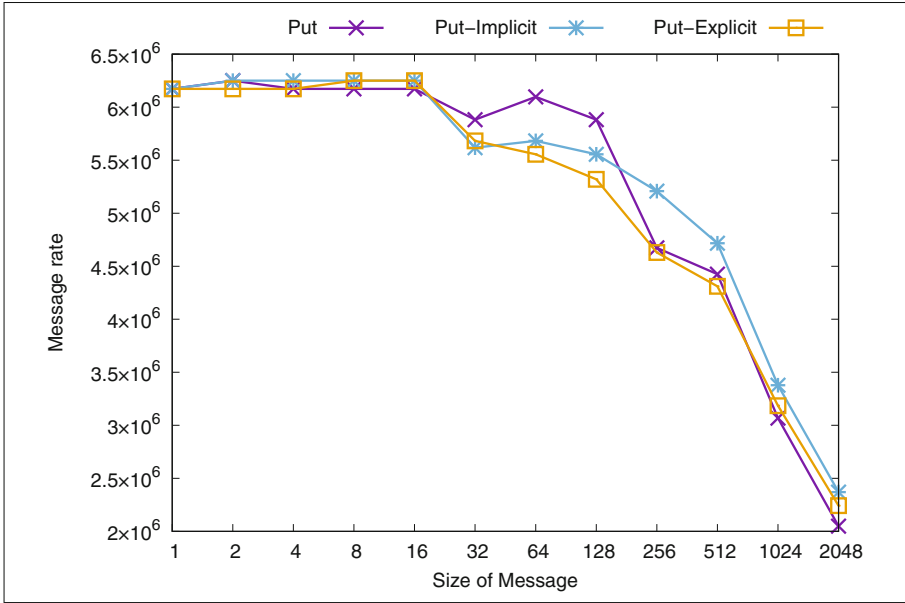
**Fig. 5.** Roundtrip latency using put-based ping-pong benchmark for large messages

For our experiments we use Mellanox’s InfiniBand HCA as network and use RC protocol for data transfer, which guarantees in-order delivery of messages. For this setup, polling on the last byte of data to learn the completion is a reasonable approach, although it might be inaccurate for networks and memory architectures that do not guarantee in-order delivery of messages. For completion, the *shm\_put* and *shm\_put\_nbi* calls require a *shm\_quiet*, while *shm\_put\_nbe* requires a *shm\_wait\_req* on the request.

From the graphs, one can observe that there are some performance differences. For a one byte message, the round trip latencies of *shm\_put*, *shm\_put\_nbi*, and *shm\_put\_nbe* are 1.58  $\mu$ sec, 1.54  $\mu$ sec, and 1.52  $\mu$ sec respectively. For 4 MB message, the latencies are 753.29  $\mu$ sec, 704.54  $\mu$ sec, and 685.65  $\mu$ sec respectively. The performance difference in case of small message is negligible.

### 5.2.3 Message Rate Evaluation

To understand the impact on message rate, we measure the message rate achievable using various *Put* interfaces. Figure 6 shows the message rate of *shm\_put\_nbe*, *shm\_put\_nbi* and *shm\_put*. For this experiment, we modify and use the message rate benchmark in OSU benchmark suite [15]. To measure the message rate of *shm\_put*, the benchmark issues a series of *Put* operations in a loop and single quiet operation at the end of the loop. Similarly, the modified benchmark for implicit *Put* issues a series of *shm\_put\_nbi* operations and single quiet operation to measure the message rate of implicit non-blocking operations. For the non-blocking *Put* with requests, the benchmark issues a *shm\_put\_nbe* and complete it with *shm\_wait\_nb* (Box 3) operation. So, it



**Fig. 6.** Comparing the message rates of *shmem\_putmem*, *shmem\_putmem\_nbi* and *shmem\_putmem\_nbe*

issues one *shmem\_wait\_nbe* operation per *shmem\_put\_nbe*. In the Fig. 6, we can observe that the message rate of RMA operations with requests is similar to blocking and implicit non-blocking *Put* operations.

### 5.3 Performance Evaluation with Scalable Synthetic Compact Applications (SSCA) #1 Kernel

**SSCA #1 Description:** The benchmark is an implementation of the Smith-Waterman local sequence alignment algorithm [2]. For our experiments, we use the OpenSHMEM version ported by Baker et al. [3]. This benchmark focuses on sequence alignment algorithms in computational biology. It stresses integer and character operations, and requires no floating point operations.

**Listing 1.1.** SSCA#1 Kernel 1 original source-code

```

1  get_data() {
2      previous_match = get(A, i-1, j-1);
3      main_codon = get(main_codon_seq, i);
4      match_codon = get(match_codon_seq, j);
5      gap_main = get(E, i-1, j);
6      gap_match = get(F, i, j-1);
7  }

9  put_data() {
10     put(A, i, j, new_score);
11     put(E, i, j, max(new_gap_score, extend_main_gap));
12     put(F, i, j, max(new_gap_score, extend_match_gap));

```

```

13 }

15 local_align(main_codon_seq , match_codon_seq){
16     /* A is the score Matrix */
17     A[len(main_codon)][len(match_codon)];
18     /* E is the main gap matrix */
19     E[len(main_codon)][len(match_codon)];
20     /* F is the match gap matrix */
21     F[len(main_codon)][len(match_codon)];
22     /* outer loop */
23     for(outer=0; outer < 2 * length(main_codon_seq)){
24         barrier_all();
25         start = compute_local_start_index(outer);
26         end = compute_local_end_index(outer);
27         /* inner loop */
28         for(inner = start; inner < end){
29             i = compute_main_index(outer, inner);
30             j = compute_match_index(outer, inner);

32             /* blocking gets */
33             get_data();

35             new_match = sim(main,codon);
36             new_score = max(new_match , gap_main , gap_match , 0);
37             if(is_score_good(new_score)){
38                 add_new_pair(new_score , i, j);
39             }
40             new_gap_score = new_match - new_gap_penalty;
41             extend_main_gap = gap_main - extend_gap_penalty;
42             extend_match_gap = gap_match - extend_gap_penalty;

44             put_data();
45         }
46     }
47 }

```

Our work focuses on improving kernel 1 of the SSCA1 benchmark using the proposed semantics for explicit requests in OpenSHMEM as described in Sect. 3. Listing 1.1 shows the source code for Kernel 1. The main kernel is comprised of two loops. The outer loop computes the bounds for the inner loop, and the inner loop computes the scores and gaps for the current iteration. At the end of each iteration the score and gap values are updated. These values are not needed until the algorithm enters the next iteration of the outer loop.

From the message characteristics perspective, the inner loop issues *Get* and *Put* operations. The *Get* operations are completed before the start of next iteration, and the *Put* operations can be completed to after all iterations of the inner loop are completed.

The first experiment looks at improving the performance by replacing the *Put* operations to update the score and gap values at the end of the inner loop with non-blocking operations. Since the algorithm employs a barrier at the beginning of the outer loop, and the barrier completes all outstanding operations, implicit non-blocking operations are used (see Listing 1.3).

For the second set of experiments, we improve the benchmark by using explicit non-blocking operations for the prefetch operations in the inner loop. This removes the requirement to issue a *shmem\_quiet* call, but uses a *shmem\_wait\_req* call on outstanding operation instead (see Listing 1.4).

**Listing 1.2.** SSCA1 with prefetching (ssca1-prefetch)

```

1  get_data() {
2      nb_previous_match = get_nb(A, nb_i-1, nb_j-1);
3      nb_main_codon = get_nb(main_codon_seq, nb_i);
4      nb_match_codon = get_nb(match_codon_seq, nb_j);
5      nb_gap_main = get_nb(E, nb_i-1, nb_j);
6      nb_gap_match = get_nb(F, nb_i, nb_j-1);
7  }

14 wait_for_previous_gets() {
15     shmem_quiet();
16 }

18 local_align(main_codon_seq , match_codon_seq){
    ...

32     /* prestage non-blocking operations */
33     get_data()

35     /* inner loop */
36     for(inner = start; inner < end){
37         i = compute_main_index(outer, inner);
38         j = compute_match_index(outer, inner);

40         nb_i = compute_next_main_index(outer , inner);
41         nb_j = compute_next_match_index(outer , inner);

43         wait_for_previous_gets();
44         previous_match = nb_previous_match;
45         main_codon = nb_main_codon;
46         match_codon = nb_match_codon;
47         gap_main = nb_gap_main;
48         gap_match = nb_gap_match;

        ...
64     }

```

The last experiment uses the interfaces in Box 5 to merge the requests. Instead of keeping track of multiple outstanding operations, operations that are dependent use a merged request (see Listing 1.5). Thus improving the usability and simplifying the program. Furthermore, there are fewer calls into the OpenSHMEM library, since there are fewer requests to wait for. Additionally an OpenSHMEM library implementation could employ optimizations to improve the performance by completing the requests in batches.

**Performance:** Figure 7 shows the results of running the benchmark on 16 nodes and with an increasing number of processes per node. The various implementations used in the experiment are as follows: *SSCA1* is the original implementation [3]. The *prefetch* is the original implementation with prefetch enabled, and with implicit *Get* operations. The *prefetch-nbi* is a modification to implementation to use implicit non-blocking RMA operations. The *prefetch-explicit* is a modified version using RMA operations with requests, and *prefetch-merged* is a modified version using RMA operations with one request for multiple *Put* operations.

**Listing 1.3.** SSCA1 with non-blocking puts (ssca1-nbi)

```

9  put_data() {
10     put_nbi(A,i,j,new_score);
11     put_nbi(E,i,j,max(new_gap_score, extend_main_gap);
12     put_nbi(F,i,j,max(new_gap_score, extend_match_gap);
13 }

```

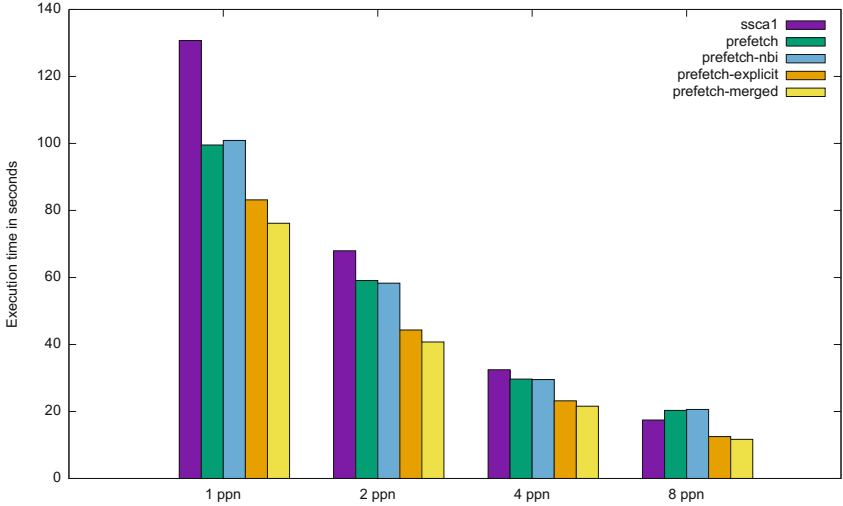
**Listing 1.4.** SSCA1 with explicit non-blocking operations (ssca1-explicit)

```

1  get_data() {
2     nb_previous_match = get_nbe(A, nb_i-1, nb_j-1, req1);
3     nb_main_codon = get_nbe(main_codon_seq, nb_i, req2);
4     nb_match_codon = get_nbe(match_codon_seq, nb_j, req3);
5     nb_gap_main = get_nbe(E, nb_i-1, nb_j, req4);
6     nb_gap_match = get_nbe(F, nb_i, nb_j-1, req5);
7  }

14 wait_for_previous_gets() {
15     shmem_wait_req(req1);
16     shmem_wait_req(req2);
17     shmem_wait_req(req3);
18     shmem_wait_req(req4);
19     shmem_wait_req(req5);
20 }

```

**Fig. 7.** Comparing performance of *ssca1* on 16 nodes

From Fig. 7, we observe the performance of the implementation using RMA operations with requests and merging of requests outperforms the original implementation and implementation with implicit RMA operations. For 16 nodes with one PE per node, the RMA operations with explicit requests outperforms the original implementation by 72% and the version with prefetch enabled by 31%. Similarly, for 128 PEs (16 nodes with 8ppn) it outperforms the original

**Listing 1.5.** SSCA1 with explicit non-blocking operations and merged requests (ssca1-merged)

```

1  get_data() {
2      nb_previous_match = get_nbe(A, nb_i-1, nb_j-1, req);
3      nb_main_codon = get_nbe(main_codon_seq, nb_i, req);
4      nb_match_codon = get_nbe(match_codon_seq, nb_j, req);
5      nb_gap_main = get_nbe(E, nb_i-1, nb_j, req);
6      nb_gap_match = get_nbe(F, nb_i, nb_j-1, req);
7  }

      wait_for_previous_gets() {
14     shmем_wait_req(req);
15 }

```

implementation by 49% and the prefetching version 74% (Note, that for 128 PEs the original version is outperforming the prefetching version).

## 6 Discussion

In this paper we introduce RMA operations with explicit requests. Since each operation can be tracked with an explicit request, an OpenSHMEM user can have a fine grained control over these operations. The consequence of this semantic is the overhead of creating and managing explicit requests for each of these operations. Our hypothesis is that with sound design and implementation, these costs can be hidden, and the impact can be mitigated. Also, from our experience in implementing network layers, we believe that for many networks it is required to manage some network descriptor at the network driver level, so exposing this to the user adds only negligible overhead. To demonstrate this, we implemented these interfaces and systematically evaluated the performance impact with micro-benchmarks and application kernels.

Our results demonstrate that a well designed and implemented OpenSHMEM stack can hide performance overhead of allocating and managing explicit handles. From Fig. 3, we can observe the performance advantages of using RMA operations with explicit requests for some communication patterns where completion of operation is not required immediately. From Figs. 4 and 5, we observe that latency of *Get* and *Put* operations with and without handles are similar. From the Fig. 6, we observe that the impact on the message rate is minimal.

In addition to RMA operations with explicit requests, we introduce the semantics of merging these requests. This can enhance the productivity and simplify some of OpenSHMEM programs as seen in modifying the SSCA #1 benchmark kernel in Sect. 5.3. Further, we see that rewriting the kernel using RMA operations with explicit requests and merging of requests can have performance benefits as seen in Fig. 7. From our investigation, we can attribute the performance advantages to the local completions used by RMA operations with explicit requests. In this case, we only flush the endpoints which exchange the messages. Further, in the case where we merge our requests, we complete the requests in a batch. On the contrary, in the case of RMA operations with no handles, all endpoints have to be flushed resulting in a higher overhead.

## 7 Related Work

Non-blocking communication is not a new concept. MPI [6] implementations of non-blocking message passing have been discussed since 2003. In the MPI-1 programming model non-blocking operations were realized through *MPI\_Isend*, *MPI\_Irecv*, *MPI\_Wait*, and *MPI\_Test*. The non-blocking communication is accomplished by the sending process issuing a *MPI\_Isend* and immediately returning to continue executing unrelated work, the receiving process would simultaneously issue an *MPI\_Irecv* and overlap this with other computations till the requested data was actually required. Completion of a data transfer can be tested through *MPI\_Test* and waited on till completion through *MPI\_Wait*.

A number of studies have compared the different non-blocking implementations of the MPI Standard [11, 13, 16]. The implementations are largely dependent on the underlying implementation and hardware support. Non-blocking collectives have also been discussed and implemented in MPI-2 [8–10]. Many large scale scientific applications like simulation of seismic wave propagation [12] and parallel FDTD algorithm [7] have benefited from non-blocking communication operations.

## 8 Future Work

In the near future, we plan to implement an OpenSHMEM library that can safely invoke OpenSHMEM interfaces from multiple user threads using RMA operations with explicit requests. Then, we plan to implement and mimic implementation of the Context proposal [5] using RMA operations with merged requests. Also, we plan to explore and characterize the application communication characteristics that can take advantage of fine grained control and completion of RMA operations.

**Acknowledgments.** This work is supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center located at the Oak Ridge National Laboratory.

## References

1. OpenSHMEM specification 1.3. [http://openshmem.org/site/sites/default/site\\_files/OpenSHMEM-1.3.pdf](http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.3.pdf)
2. Bader, D., Madduri, K., Gilbert, J., Shah, V., Kepner, J., Meuse, T., Krishnamurthy, A.: Designing scalable synthetic compact applications for benchmarking high productivity computing systems (2006)
3. Baker, M., Welch, A., Gorentla Venkata, M.: Parallelizing the Smith-Waterman algorithm using OpenSHMEM and MPI-3 one-sided interfaces. In: Gorentla Venkata, M., Shamis, P., Imam, N., Lopez, M.G. (eds.) OpenSHMEM 2014. LNCS, vol. 9397, pp. 178–191. Springer, Heidelberg (2015). doi:[10.1007/978-3-319-26428-8\\_12](https://doi.org/10.1007/978-3-319-26428-8_12)



4. ten Bruggencate, M., Roweth, D., Oyanagi, S.: Thread-safe SHMEM extensions. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 178–185. Springer, Heidelberg (2014). doi:[10.1007/978-3-319-05215-1\\_13](https://doi.org/10.1007/978-3-319-05215-1_13)
5. 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, PGAS 2014, NY, USA, pp. 10:1–10:9. ACM, New York (2014). <http://doi.acm.org/10.1145/2676870.2676872>
6. Dongarra, J.J., Otto, S.W., Snir, M., Walker, D.: An Introduction to the MPI Standard, University of Tennessee, Knoxville, TN, USA (1995). [http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai%3Ancstrlh%3Autk\\_cs%3Ancstrl.utk\\_cs%2F%2FUT-CS-95-274](http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai%3Ancstrlh%3Autk_cs%3Ancstrl.utk_cs%2F%2FUT-CS-95-274)
7. Guiffaut, C., Mahdjoubi, K.: A parallel FDTD algorithm using the MPI library. *IEEE Antennas Propag. Mag.* **43**(2), 94–103 (2001)
8. Hoefler, T., Kambadur, P., Graham, R.L., Shipman, G., Lumsdaine, A.: A case for standard non-blocking collective operations. In: Cappello, F., Herault, T., Dongarra, J. (eds.) EuroPVM/MPI 2007. LNCS, vol. 4757, pp. 125–134. Springer, Heidelberg (2007). doi:[10.1007/978-3-540-75416-9\\_22](https://doi.org/10.1007/978-3-540-75416-9_22)
9. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and performance analysis of non-blocking collective operations for MPI. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC 2007, pp. 1–10. IEEE (2007)
10. Hoefler, T., Squyres, J., Bosilca, G., Fagg, G., Lumsdaine, A., Rehm, W.: Non-blocking collective operations for MPI-2. Open Systems Lab, Indiana University, Technical report 8 (2006)
11. Liu, J., Chandrasekaran, B., Wu, J., Jiang, W., Kini, S., Yu, W., Buntinas, D., Wyckoff, P., Panda, D.K.: Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics. In: Supercomputing, 2003 ACM/IEEE Conference, pp. 58–58. IEEE (2003)
12. Martin, R., Komatitsch, D., Blitz, C., Goff, N.: Simulation of seismic wave propagation in an asteroid based upon an unstructured MPI spectral-element method: blocking and non-blocking communication strategies. In: Palma, J.M.L.M., Amestoy, P.R., Daydé, M., Mattoso, M., Lopes, J.C. (eds.) VECPAR 2008. LNCS, vol. 5336, pp. 350–363. Springer, Heidelberg (2008). doi:[10.1007/978-3-540-92859-1\\_32](https://doi.org/10.1007/978-3-540-92859-1_32)
13. Saif, T., Parashar, M.: Understanding the behavior and performance of non-blocking communications in MPI. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 173–182. Springer, Heidelberg (2004). doi:[10.1007/978-3-540-27866-5\\_22](https://doi.org/10.1007/978-3-540-27866-5_22)
14. Shamis, P., Venkata, M.G., Lopez, M.G., Baker, M.B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R.L., Liss, L., Shahar, Y., Potluri, S., Rossetti, D., Becker, D., Poole, D., Lamb, C., Kumar, S., Stunkel, C., Bosilca, G., Bouteiller, A.: UCX: an open source framework for HPC network APIs and beyond. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 40–43, August 2015
15. The Ohio State University: OSU micro-benchmarks (2016). <http://mvapich.cse.ohio-state.edu/benchmarks/>
16. Tipparaju, V., Krishnan, M., Nieplocha, J., Santhanaraman, G., Panda, D.: Exploiting non-blocking remote memory access communication in scientific benchmarks. In: Pinkston, T.M., Prasanna, V.K. (eds.) HiPC 2003. LNCS, vol. 2913, pp. 248–258. Springer, Heidelberg (2003). doi:[10.1007/978-3-540-24596-4\\_27](https://doi.org/10.1007/978-3-540-24596-4_27)

OpenSHMEM and Related Technologies. Enhancing  
OpenSHMEM for Hybrid Environments  
Third Workshop, OpenSHMEM 2016, Baltimore, MD, USA,  
August 2 – 4, 2016, Revised Selected Papers  
Gorentla Venkata, M.; Imam, N.; Pophale, S.; Mintz, T.M.  
(Eds.)  
2016, X, 239 p. 102 illus., Softcover  
ISBN: 978-3-319-50994-5