Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization

Roberto Belli, Torsten Hoefler
Dept. of Computer Science, ETH Zurich
{bellir,htor}@inf.ethz.ch

***Abstract*—Remote Memory Access (RMA) programming enables direct access to low-level hardware features to achieve high performance for distributed-memory programs. However, the design of RMA programming schemes focuses on the memory access and less on process synchronization. For example, in contemporary RMA programming systems, the widely used producer-consumer pattern can only be implemented inefficiently, incurring the overhead of an additional round-trip message. We propose Notified Access, a scheme where the target process of an access can receive a completion notification. This scheme enables direct and efficient synchronization with a minimum number of messages. We implement our scheme in an open source MPI-3 RMA library and demonstrate lower overheads (two cache misses) than other point-to-point synchronization mechanisms. We also evaluate our implementation on three real-world benchmarks: a stencil computation, a tree computation, and a Cholesky factorization implemented with tasks. Our scheme always performs better than traditional message passing and other existing RMA synchronization schemes, providing up to 50% speedup on small messages. Our analysis shows that Notified Access is a valuable primitive for any RMA system. Furthermore, we provide guidance for the design of low-level network interfaces to support Notified Access efficiently.**

***Keywords*-RMA; synchronization; notification; MPI**

I. Introduction

Modern high-performance networks support Remote Direct Memory Access (RDMA) which allows processes to directly access userspace memory of any remote process bypassing the remote CPU and operating system. Traditionally, this direct access has been used to implement fast message passing systems [1]. However, the semantics of message passing require expensive interactions at the receiving side where an incoming message needs to be matched to the correct receive statement in order to determine the buffer address for the message. Emerging RMA and PGAS programming models enable the programmer to specify the target buffer at the source process which allows RDMA hardware to perform the whole transmission without software interaction. This often leads to significant improvements in application performance [2]–[4].

Most current RMA programming models, such as UPC, Fortran 2008, SHMEM, or MPI-3 One Sided, mainly focus on data movement (e.g., `put` and `get`) and memory synchronization (e.g., `flush` or `fence`). Process synchronization is often performed via non-scalable bulk synchronization functions (e.g., `barrier` in UPC and SHMEM; `fence` in MPI-3 One Sided; `sync all` in Fortran 2008), scalable group synchronization functions (e.g., `post`, `start`, `complete`, `wait` in MPI-3 One Sided; `sync images` in Fortran 2008), or through busy waiting on memory locations (in combination with memory synchronizations such as `flush` in MPI-3 or `sync memory` in Fortran 2008).

However, as we will show, each of these synchronization mechanisms requires at least one additional round-trip message between the processes. This synchronization overhead may drastically reduce the performance of applications where a processor is forced to stall until the synchronization is acknowledged. For instance, this overhead is critical in the bounded buffer form of the general producer-consumer pattern. This pattern is common in parallel applications using any form of halo exchange, tree-based communication, or any tasking system. Such producer-consumer patterns are ideally supported by message passing where the completion of the receive indicates the reception of the message.

In this work, we combine the advantages of RMA programming—hardware-supported direct memory access—with the process synchronization features of message passing. We propose to extend RMA programming models with a new mechanism called *Notified Access* which allows the target process to detect when a transfer is completed without additional messages. Figure 1 shows the performance of a strong scaling stencil computation with various numbers of processes [5]. As expected, message passing performs better than One Sided approaches due to additional round-trip times. Notified Access consistently outperforms message passing by more than 1.4x on 32 processes.

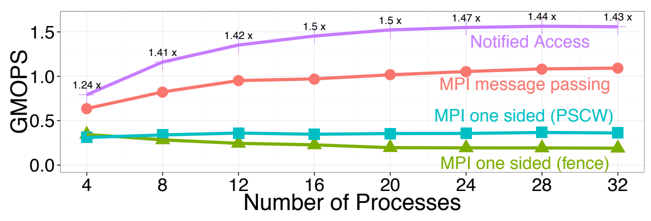
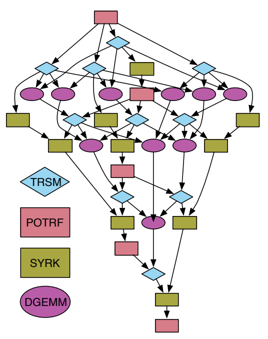
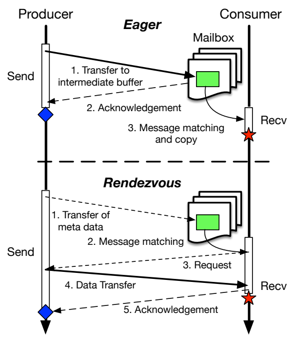


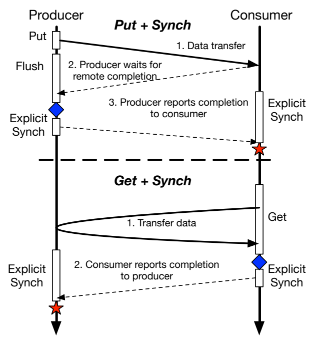
Figure 1: Pipeline stencil performance in billion memory operations per second (GMOPS) of various communication schemes on a domain of size 1280x12800 [5].



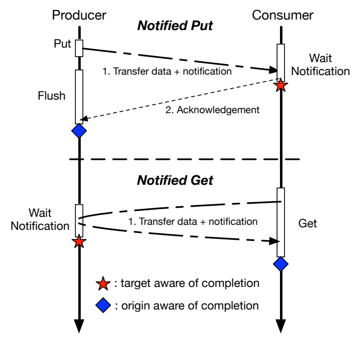
(a) Cholesky task graph



(b) Message Passing



(c) RMA


Producer
Consumer
Notified Put
Put
Wait Notification
1. Transfer data + notification
Flush
2. Acknowledgement
Notified Get
Wait Notification
1. Transfer data + notification
Get
★ : target aware of completion
◆ : origin aware of completion

(d) Notified Access

Figure 2: An example task graph illustrating a producer-consumer problem and examples of various point-to-point protocols to implement producer-consumer communications. The point in time at which a process knows that the transfer is complete is denoted for the target and the origin processes as a star and a diamond, respectively.

Notified Access creates a programming model with semantics in between pure message passing and pure (passive) RMA programming, combining the strengths of both.

The main contributions of our work are:

- We propose *Notified Access* as a synchronization model that extends RMA programming with per-message process synchronization and matching.
- We describe a small extension to the MPI specification that enables notified access semantics. Similar extensions can be added to other RMA programming models.
- We discuss how to implement notified access with minimal overhead on various existing low-level network interfaces and we develop a fully-functional implementation for Cray systems.
- We present three case-studies: a stencil computation with a 3-point stencil, a tree computation, and a statically scheduled tasking system to show the benefits of notified access in practice.

II. Background

RMA programming is becoming an important tool for writing fast parallel programs. However, as hinted before, many RMA models were designed to exploit existing hardware capabilities and to support billboard-style [6, §11] random-access applications. The direct hardware acceleration in modern networks enables a much wider set of use-cases for RMA programming that are not well supported by existing RMA or PGAS models. In this paper, we focus on investigating the ubiquitous producer-consumer pattern in parallel computing.

Producer-consumer communications are found in virtually all modern high-performance applications. They are employed whenever a dataflow needs to cross domain (process boundaries). In the general form of the problem, a producer generates data and enqueues it for the consumer. The consumer then dequeues the data and uses it to advance the computation. For example, Figure 2a shows the task graph induced by a Cholesky factorization. Each edge in the graph represents a producer-consumer relation. We will explain the Cholesky example in more detail in Section VI-C. Since queuing is expensive, most high-performance codes implement a single-element queue in the form of a receive buffer at the destination.

All producer-consumer communications require two basic semantics: (1) data transmission and (2) process synchronization. In message passing, both semantics are provided by the receive. Figure 2b shows the network transactions required to implement a transfer of a small (eager) and a large (rendezvous) message [7]. RMA programming models separate data transfer and synchronization into different primitives. The progress of many data transfers can then be synchronized with a single (bulk) operation.

However, many producer-consumer synchronizations only require a single message and the needed synchronization implies additional network transactions. Figure 2c shows protocols based on remote put and get, respectively. All protocols, with the exception of eager message passing, require at least three message transactions on the critical path. Eager message passing only requires a single transaction but the receiver needs to match and copy incoming messages, which causes different overheads. In addition, the required intermediate buffers make eager message passing generally unscalable [7].

Notified access provides the option to unify these two semantics, enabling pipelining and bulk synchronization with single-transfer notification. Figure 2d shows notified access for small as well as large message sizes. Notified access only requires a single message for the actual network transmission and the notification and enables full asynchronous message progression in hardware. We will describe details in Section III.

A. *Notation*

We now briefly introduce the terminology we use in the remainder of the document. In RMA, we define the *origin* process as the process that issues the RMA operation and the *target* process as the process that the RMA operation targets. We define a *remote access* as an RMA operation that copies an ordered set of (not necessarily consecutive) bytes either from the origin to the target (put) or vice-versa (get). Some more complex atomic accesses involve get and put in a single operation (e.g., compare and swap).

A *synchronization epoch* [6, §11] is a time interval relative to an origin and a target process. Each epoch is enclosed by two synchronization operations that are called at the origin and directed towards the target process. Two remote accesses (between the same origin-target pair) are not ordered if they happen in the same epoch. However, if they appear in different epochs, then they are ordered by the order of the epochs. The term *notification* represents the transmission of the information that an epoch has ended to the target process. The end of the epoch indicates that the target process can access the communicated data or re-use its local communication buffers. In most RMA models, the target process is passive and is not informed of accesses to its memory. Thus, in most of today's RMA models, producer-consumer computations require an additional round-trip message to communicate the change of synchronization epochs.

III. Notified Access

Notified Access adds a remote completion notification to any remote access. The target process can use this notification for synchronizing local or remote accesses to the buffer. The interpretation of the notification depends on the action: if the notified access is a read then the notification indicates that the data was copied and the buffer can be overwritten; if the notified access is a write then the notification indicates that the data was committed to memory and can be read.

The origin can mark accesses with a notification or without, i.e., not all accesses have to trigger a remote notification. The simplest notification system would just notify the target process of each incoming notified access. However, this would not make it possible to distinguish different accesses at the target. Thus, we propose a richer interface that allows the target to set up notifications that *match* a specific origin and an arbitrary integer tag value. The resulting matching queue semantics have been highly successful in MPI-1.

The <source, tag> tuple can be used to identify specific accesses in the program logic. The notification system also supports wildcards for both source or tag which match incoming messages in the order of arrival to the oldest notification if multiple notifications match. The tag can be selected to identify accessed memory regions at the target and can thus be used to efficiently implement starvation-free dataflow-based tasking systems. Since RMA programming often involves many small accesses, we introduce counting notifications that only notify after n matching accesses were performed. This capability allows bulk-notification optimizations.

Another option for remote notification would be to define notified synchronization operations instead of notified accesses, e.g., a notified version of the common `flush` operation. However, this would still necessitate at least two network transfers (which could be pipelined) to transmit a single message in a producer-consumer setting. Notified access only requires a single network transfer. To achieve similar semantics and maintain the epoch definition, one could consider that a notified access has the side-effect of completing all previous accesses. This would enable an elegant definition of epoch which is enclosed by notified accesses or synchronizations. However, we do not consider this because no network technology supports this today and it seems hard to guarantee this without additional network transfers on adaptively routed networks. If this becomes available in the future then one may easily redefine the epoch concept. In our definition of notified access, all notified accesses form their own epoch and do not interact with normal remote accesses.

A. *Discussion: Shared Memory Synchronization*

We now briefly draw some parallels between RMA programming and shared memory programming. Both models are very similar in that remote memory can be accessed directly and data is moved by the hardware. The main difference is that RMA programming presents a partitioned view of the address space (thus also often called Partitioned Global Address Space, PGAS, even though a PGAS can be implemented without direct remote access primitives).

A shared memory synchronization is performed through memory fence ISA instructions (e.g., `mfence` in x86). These instructions typically block all further instructions until all write buffers are empty and all accesses are visible to all cores. Such fences are semantically similar to flush operations in RMA. However, flush operations synchronize across datacenters where latencies can be in the microsecond range (as opposed to nanoseconds for on-chip transfers). Thus, frequent flushes have a detrimental effect on application performance. Existing RMA and the new (counted) notified access bulk synchronization operations enables efficient message pipelining for high-latency environments.

We believe that Notified Access may also be a viable interface for future large-scale on-chip networks where transfer pipelining becomes a must and synchronization has a higher relative cost.

B. *A Strawman Interface for Notified Access*

While Notified Access is independent of a particular programming model, we will now present a strawman interface for the Message Passing Interface (MPI). Without loss of generality, we will use this interface as a case study to compare applications using a highly-tuned MPI-3.0 [6] implementation. We chose MPI for several reasons: First, it has the richest combination of synchronization mechanisms—all synchronization mechanisms provided by other models can be considered as a subset of those provided by MPI [6]. Second, many different implementations of the MPI specification are available, optimized and fully exploit the performance of the target architecture.

To extend MPI with Notified Access, we first introduce a notified variant for each communication operation in MPI RMA. Each new function has an additional integer tag argument. The following listing shows C interfaces for put and get[1]:

```
int MPI_Put_notify(void *origin_addr, int origin_count,
                   MPI_Datatype origin_type, int target_rank,
                   MPI_Aint target_disp, int target_count,
                   MPI_Datatype target_type, MPI_Win win, int tag);
int MPI_Get_notify(void *origin_addr, int origin_count,
                   MPI_Datatype origin_type, int target_rank,
                   MPI_Aint target_disp, int target_count,
                   MPI_Datatype target_type, MPI_Win win, int tag);
```

Similar functions can be created for MPI's accumulate operations (accumulate, get accumulate, fetch and op, and compare and swap) or request-based operations. As in other MPI routines, the number of significant tag bits may be limited due to hardware constraints. The calls support zero-byte payloads in which case only the notification is set.

MPI request objects are used for notification at the receiver side. In order to keep the overhead minimal, we use persistent requests [6, §3.9]. These requests are initialized explicitly with the function MPI_Notify_init and are not automatically freed. We show the C interface for notify init and the existing start, test, and wait functions in the following:

```
int MPI_Notify_init(MPI_Win win, int src_rank, int tag,
                    int expected_count, MPI_Request *request);
/*Functions already available in MPI*/
int MPI_Start(MPI_Request *request);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

MPI_Notify_init initializes a request for notification and binds it to a specific MPI RMA window with notification count, tag, and source. The returned MPI request object can be used with the usual MPI test and wait functions. A request completes after expected_count matching notified accesses have been performed. Matching is performed in order and it is defined through source and tag and the wildcards MPI_ANY_SOURCE and MPI_ANY_TAG are supported. If a request is completed, the returned MPI status object includes the information of only the last matching notified access. Probe semantics can be added trivially.

1) Persistent Requests: Standard MPI requests are allocated during the initialization call (e.g., nonblocking send or receive) and freed during the completing call (e.g., test or wait). This implicit request management makes these objects rather expensive. A special kind of requests, *persistent* requests has been introduced to statically bind message passing arguments (buffers and counts) to a request that can then be started and completed multiple times. Re-using requests in this manner amortizes their creation time and enables explicit request management (allocating and freeing). In Notified Access, requests are initialized with MPI_Notify_init and freed with MPI_Request_free. Before each use, requests have to be initialized with MPI_Start. After each initialization, request completion can be tested with the normal test and wait operations.

[1]While we describe MPI interfaces here for readability, our implementation uses the foMPI prefix to not violate the standardized MPI namespace

Listing 1 shows a complete example: a ping-pong benchmark to illustrate message transmission whose performance we measure in the following section.

```
MPI_Win win;
MPI_Request notification_request;
MPI_Status notification_status;
int win_size = 2 * MAX_SIZE * sizeof(double);
double *buf; int my_rank;
MPI_Win_allocate(win_size, sizeof(double), MPI_INFO_NULL,
      MPI_COMM_WORLD, &buf, &win);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
/* initialize notification request */
int customTag = 99; int expected_count = 1;
MPI_Notify_init(win, partner_rank, customTag, expected_count,
      &notification_request);
for(size=8; size<MAX_SIZE; size++) {
  if (my_rank==client_rank) {
    /* send ping */
    MPI_Put_notify(buf, size, MPI_DOUBLE, partner_rank, 0, size,
      MPI_DOUBLE, win, customTag);
    MPI_Win_flush(partner_rank,win);
    /* wait for pong */
    MPI_Start(&notification_request);
    MPI_Wait(&notification_request, &notification_status);
  } else { /* server */
    /* wait for ping */
    MPI_Start(&notification_request);
    MPI_Wait(&notification_request, &notification_status);
    /* send pong */
    MPI_Put_notify(buf, size, MPI_DOUBLE, partner_rank, MAX_SIZE,
      size, MPI_DOUBLE, win, customTag);
    MPI_Win_flush(partner_rank,win);
  }
} /* end of iterations */
MPI_Request_free(&notification_request);
MPI_Win_free(&win);
```

Listing 1: A simple ping-pong example in Notified Access.

IV. Implementation

We now explain how to implement Notified Access with various modern network technologies.

Today's networks do not support hardware message matching, therefore, we implement the matching in software at the receiver. This matching mechanism does not reduce the performance significantly since the data movement is still **fully** performed in hardware. Only the processing of the light-weight notification is done in software. All software functionalities can be implemented in the test or wait functions which form synchronization points at the target and does thus not require any asynchronous activity. We will describe how our protocol requires only two compulsory cache misses in the worst case if less than four notifications are active.

Existing MPI matching protocols must either copy the whole message in the eager case or perform extra synchronization using software protocols in the rendezvous case. The expensive eager message copy pollutes the cache and consumes energy for the data movement. The rendezvous protocol either prevents asynchronous progression or requires an asynchronous software agent at the receive side [8]. Thus, we expect significant performance benefits compared to message passing implementations.

A. *Network-specific considerations*

Various RDMA networks can support Notified Access immediately. We will now briefly discuss how it can be implemented with lowest overheads using widely known network technologies.

InfiniBand: The Open Fabrics Enterprise Distribution (OFED) defines RDMA write with immediate which generates a completion queue entry at the receiver. The receiver can check this queue for remote notifications. The immediate value can be used to encode the tag. RDMA read with immediate is unfortunately not available and a more complex protocol has to be employed. For instance, one could rely on the ordering guarantees of InfiniBand and inject a notification message right after the read.

Portals 4: The Portals interface [9] provides Event Queues that are used to log performed network operations. These queues can be polled at the target in order to retrieve which remotely initiated events have been locally committed. This characteristic, combined with Portals' capability to transfer out-of-band data in the header, can be leveraged to distinguish between notified and normal operations and transfer the tag value and the data at the same time. The use of Event Queues can potentially limit the message rate in comparison to Portals counters or other mechanisms. A more detailed discussion of design tradeoffs can be found in the discussion section.

Other network interfaces have similar primitives that can be used to implement Notified Access semantics. We now present a complete blue print implementation for Cray networks as well as shared memory in detail.

B. *Implementation on Cray Networks*

Our implementation bases on the open source foMPI (*Fast One Sided MPI*) [4] which supports the full MPI-3.0 One Sided interface. The foMPI library uses DMAPP [10] and XPMEM [11] APIs for inter- and intra-node communications, respectively. Our extended *foMPI-NA*[2] utilized the uGNI API [12] that provides direct access to Cray's Fast Memory Access (FMA) and Block Transfer Engine (BTE) mechanisms. FMA provides a service to efficiently transfer small amounts of contiguous data and BTE targets larger transfers and offloads them to the network card. With both mechanisms is it possible to directly notify the completion of a RDMA operation to the target process. After the completion of a remote read or write, a notification can be posted to a *destination completion queue* that can be shared between different segments of exposed memory.

[2]Available at http://spcl.inf.ethz.ch/ (Research section)

The uGNI interface allows to attach a 4-byte integer to each access which is then returned in the completion queue at the destination. This functionality is similar to RDMA write with immediate but also available for read accesses. We encode the source rank and tag into the first and last two bytes, respectively. This mechanism interacts seamlessly with the existing foMPI which uses the DMAPP interface for remote accesses.

The target side utilizes a single queue called the Unexpected Queue (UQ) to maintain the order of notifications. The call `MPI_Notify_init` only allocates the persistent notification request object and attaches source, tag, and count to it. The request is a simple 32-byte structure: two 8-byte values for the window and rank, two 4-byte values for tag and a request type, and two 4-byte values for count and matched (initialized to zero).

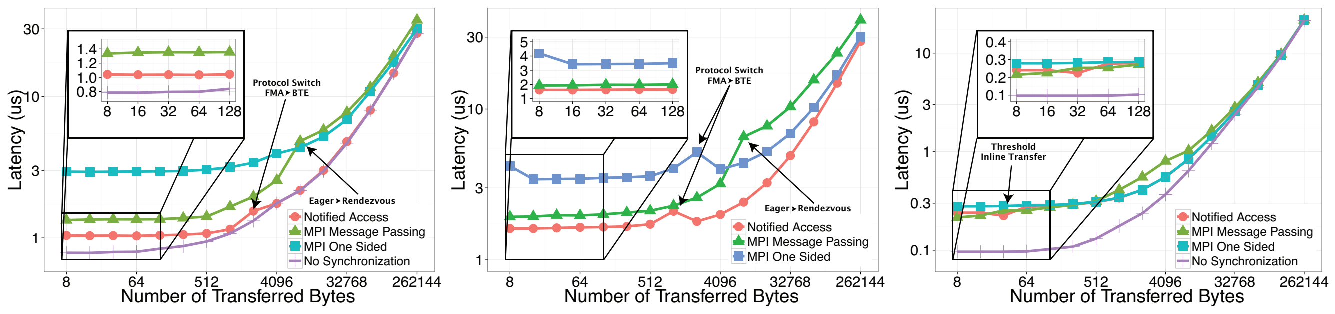
Requests are advanced only in test and wait functions. Wait is implemented as a loop until test succeeds and test performs the following steps: first it searches the UQ for matching notifications. If it finds a matching notification, it increases the `matched` counter of the request. If the request reaches the required matched count then the functions returns completion. If the UQ does not contain all the needed notifications the check continues polling the uGNI *destination completion queue* associated with the target window. While polling the completion queue, we may encounter notifications that do not match the query parameters. These notifications are appended to the UQ for later matching.

Once a notification is returned to the user, the request will remain marked as completed (`matched=count`). The function `MPI_Start` simply resets the matched counter to zero to reinitialize the request.

C. *Implementation in Shared Memory*

For shared memory, we create a bounded ring buffer for notifications in a set of shared memory segments which are shared between all processes using XPMEM. Each process owns such a segment and notifications that target that process are enqueued into the ring buffer. Each entry in the notification buffer is exactly the size of one cache line and a notification contains, in addition to source and tag, a payload field including the destination offset and the data.

The data of small put accesses can thus be directly added to the notification to reduce the number of cache line transfers. We call this protocol *inline transfer*. Larger accesses are performed using an optimized `memcpy` and a memory fence which is followed by the notification.



(a) Inter-node put (b) Inter-node get (c) Intra-node put/get

Figure 3: Ping-pong benchmark performance comparing Message Passing, MPI-3.0 One Sided, and Notified Access.

The target performs similar operations as for the uGNI implementation. In addition to the uGNI completion queue it checks the XPMEM notification queue. Non-matching notifications are appended to the same unexpected queue and matching is identical.

V. Microbenchmarks

We now briefly analyze the number of additional cache misses at the target assuming that exactly one notification is active and no structure is in cache. The first mandatory cache miss is caused by loading the 32 Byte request itself (we assume the user aligned the structure). The second miss is caused by accessing the UQ which can be arranged in a way that the first elements are always on the same cache line as the head pointer. More misses may be caused by accessing the hardware completion queues (uGNI or XPMEM). But since any notification system would incur these, we do not count them towards the overhead.

A. *Microbenchmark Evaluation and Performance Model*

We now compare the performance of Notified Access with existing MPI RMA and Message Passing mechanisms implemented in foMPI and Cray MPI. Gerstenberger at al. [4] demonstrated that foMPI outperforms other programming models such as UPC, CAF, or Cray's native MPI One Sided, thus, we use foMPI for all MPI One Sided experiments.

Firstly, we evaluate the latency and bandwidth using a standard ping-pong benchmark. The complete benchmark code for Notified Access is shown in Listing 1 and we compare the performance to traditional Message Passing, One Sided with general active target mode synchronization and fence synchronization. We also compare with an unsynchronized benchmark which indicates the lower-bound for network transmission but is not a legal synchronization implementation. Fence and general active target mode synchronizations performed identical on two processes, thus, we only report one result.

Our benchmark measures the time needed by the client process to send data to the server process and being aware that the data sent by the server is committed. Each test is repeated 1,000 times for each configuration to gather statistics. The latencies are calculated as half Round Trip Time (RTT) and all the figures show the medians of all gathered measurements.

For reproducibility and clarity, we present code fragments for each of our benchmarks. However, due to space constraints, we only present the instructions of the client process (the block in `if (my_rank==master_rank)` in Listing 1). The server performs the same operations in the opposite order and the enclosing loops are identical to Listing 1.

For MPI Message Passing we used the standard scheme that involves send and receive:

```
MPI_Send(buf, size, MPI_DOUBLE, partner_rank, tag, comm);
MPI_Recv(buf, size, MPI_DOUBLE, partner_rank, tag, comm, &status);
```

General active target synchronization is implemented as:

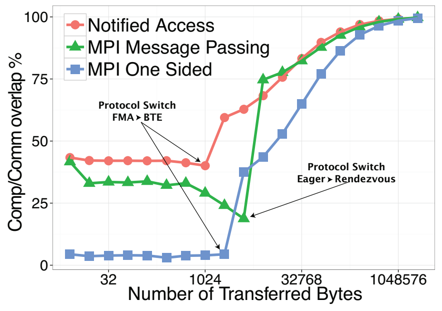
```
MPI_Win_start(group, 0, win);
MPI_Put(buf, size, MPI_DOUBLE, partner, 0, size, MPI_DOUBLE, win);
MPI_Win_complete(win);
MPI_Win_post(group, 0, win);
MPI_Win_wait(win);
```

As theoretical limit we use busy wait on the receiver side at the first and last bytes of the access (without full memory synchronization). We remark that this technique cannot be used to implement a correct (portable) program:

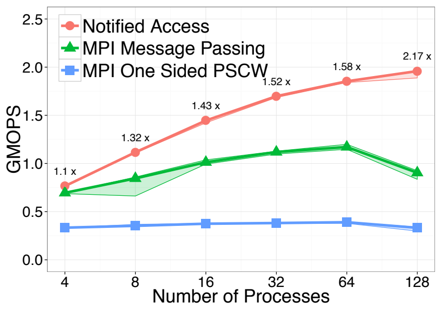
```
rcvBuf[0] = rcvBuf[size-1] = mark;
/*send and receive buffers are at different memory locations*/
MPI_Put(sndBuf, size, MPI_DOUBLE, partner, 0, size, MPI_DOUBLE, win);
MPI_Win_flush(partner,win);
while(rcvBuf[0] == mark || rcvBuf[size-1] == mark);
```

Experimental Environment: We execute all benchmarks on a Cray XC30 system at the Swiss National Supercomputing Centre (CSCS). This system, named *Piz Daint* has 5,272 compute nodes based on Intel Xeon E5 processors. Each compute node is connected to other nodes through a Dragonfly Aries network. We use the Cray Linux Environment 5.1.UP01, GCC version 4.8.2 using the O3 optimization level, cray-MPICH version 6.2.2, uGNI version 5.0-1, DMAPP 7.0.1, and XPMEM 0.1.

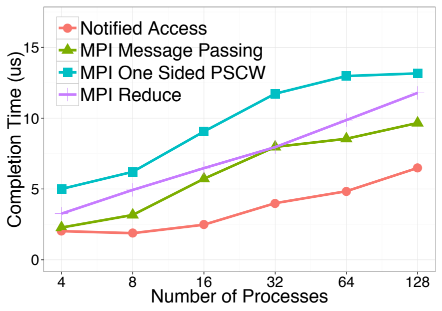
Performance: Figure 3a shows the latency for varying message sizes needed for putting data to a remote node and notifying the destination process. Notified Access implemented in foMPI-NA requires less than the 50% of the time



(a) Single-message overlap benchmark



(b) Pipelined Stencil from Intel's Parallel Research Kernels



(c) Tree-computation with 8 Bytes and 16 children at each level

Figure 4: Performance results for various application use-cases.

Table I: Varying LogGP Parameters for Notified Access

	Shared Memory	uGNI FMA	uGNI BTE
L	$0.25\mu s$	$1.02\mu s$	$1.32\mu s$
G	$0.08ns$	$0.105ns$	$0.101ns$

needed by MPI One Sided to transfer data and synchronize on small transfers. It also performs better than MPI Message Passing, which for small messages uses an optimized eager protocol incurring a buffer copy overhead. Figure 3b shows the performance of a notified get in comparison to MPI One Sided and Message Passing. We remark that the message passing performance is a single transfer and has thus an advantage over get which requires a request-reply protocol. Figure 3c shows intra-node (shared memory) results. As expected, notified access performance similar to message passing because the latency for a round-trip is negligible in shared memory where the overhead of the notification dominates for small messages.

We now proceed to model the performance of each involved call. The time to perform `MPI_Notify_init` is $t_{init} = 0.07\mu s$ and the time to perform `MPI_Request_free` is $t_{free} = 0.04\mu s$ in our implementation. Starting a request incurs setting a single integer which costs $t_{start} = 0.008\mu s$. Issuing a put or get notify costs $t_{na} = 0.29\mu s$.

We now model the transmission and remote notification time. We provide simple LogGP parameters [13] for the notified put and get for intra-node as well as inter-node communications. The send overhead is $o_s = t_{na}$ and we determine the receive overhead $o_r = 0.07\mu s$ if there is a single request in the queue. L is the zero-byte latency, and G is the transmission cost per Byte payload. The latter two parameters depend on the transport type (shared memory or uGNI) as well as the uGNI option FMA or BTE (which is selected based on message size). Table I shows L and G for all three options.

Computation/Communication Overlap: Our overlap benchmark measures what share of the communication latency can be overlapped with computation. For each data size, it calibrates the computation to consume slightly more time than the communication latency. Then, it places this computation between the communication initiation (`MPI_Isend`, `MPI_Put`, or `MPI_Put_notify`) and the local completion (`MPI_Wait`, `MPI_Win_fence`, or `MPI_Win_flush`). The ratio of communication overlap is then computed from the measured latency and the overhead of the communication (init + completion). Figure 4a shows the ratio of overlappable communication. Small messages are hard to overlap because either the notification time in fence or the message passing overheads cannot be hidden. Large messages are overlapped well in One Sided due to hardware offload. Cray message passing implementation offers asynchronous progression for the rendezvous protocol at the cost of CPU resources [8]. Notified access achieves the expected high overlap for all sizes because there are no additional notification or copy overheads and the transfer is fully hardware offloaded. The higher relative overlap of MPI Message Passing at 8KB in comparison with Notified Access is due to the higher message passing latency for that message size.

VI. Applications

We now discuss several use-cases of Notified Access: a pipelined stencil, a tree computation, and a parallel Cholesky factorization. Each of these use-cases demonstrates a different version of a producer-consumer problem where Notified Access can be used.

A. Pipelined Stencil

Our first motif application represents a pipelined stencil computation. For this, we use the `Sync_p2p` kernel from the Intel Parallel Research Kernels (PRK) [5]. We chose this benchmark because it is designed to test the efficiency of point-to-point synchronization. It represents several numerical methods, such as wavefront-parallel algorithms or Lower-Upper Symmetric Gauss-Seidel.

In the kernel, a two-dimensional $m \times n$ domain is decomposed row-wise and x columns are assigned to each process. A simple 3-point stencil update

`A(i,j)=A(i-1,j)+A(i,j-1)-A(i-1,j-1)` is performed at each point. The execution is pipelined such that the first process starts updating its x values in the first row. When this is finished, it starts sending the first-row halo to the next process and continues to compute the second row. Processes wait for the update from the left neighbor and pipeline the data to the right. All processes work in parallel as soon as the pipeline is filled. A global synchronization is performed to synchronize between iterations once all processes finish their mth (last) row of the matrix.

For our test, we port Intel's implementation to MPI One Sided using both fence and general active target mode (PSCW) as synchronization mechanisms. Figure 4b shows the performance of the various implementations in a weak scaling experiment. We show billion memory accesses per second (GMOPS) to compare the communication modes. We keep the partition size per PE constant with 1280×1280. We plot the average of ten runs and the shade shows the 99% confidence interval.

These results show that Notified Access improves the performance of the pipelined stencil more than 2.17x over Message Passing. As expected, other One Sided approaches are not suited well for this pattern and thus perform sub optimal. General active target synchronization performs better than fence because it only synchronizes two neighbors instead of all processes.

B. Hierarchical Tree Computations

Hierarchical computations are ubiquitous in parallel programs. We represent such computations by a tree-based communication which represents fan-in/fan-out as well as scatter/gather patterns. Such patterns are often used in hierarchical computations such as the Fast Multipole Method, Barnes Hut, or computations with hierarchically-structured matrices.

To represent these patterns, we implemented a 16-ary tree performing a reduction at each stage. We again implement it with Message Passing, One Sided general active target synchronization as well as Notified Access. For notified access, we use the counting feature to wait for all incoming children with a single request. Figure 4c shows the average times for small message reductions. We also compare to the semantically equivalent vendor optimized Cray `MPI_Reduce` operation. The difference is significant for latency-bound small-message transfers, where notified access even outperforms the Cray's optimized reduction.

Consumer-managed Buffering: All previous examples required the producers to manage the buffering at the consumer (i.e., the producer had to specify the target address). This management may be expensive for computations where multiple producers send data to a single consumer and the set of producers changes nondeterministically (e.g., in many dynamic applications such as particle codes or graph computations). A notified get can be used to retrieve the data from the remote processes and simplifies buffer management.

C. Task-based Cholesky Factorization

We now present a full Cholesky factorization example to demonstrate the utility of Notified Access in task dataflow settings. The real Cholesky factorization of an $n \times n$ symmetric positive-definite matrix A has the form $A = LL^T$ where L is an $n{\times}n$ real lower triangular matrix and L^T is the conjugate transpose. It is often used for the numerical solution of linear equations $Ax = b$ by the forward-substitution $Ly = b$ followed by the back-substitution $L^T x = y$. Such matrices occur frequently in numerical solutions of partial differential equations and Monte Carlo simulations.

The factorization can be calculated using different algorithmic variants: for instance the right-looking or left-looking. We use the left-looking variant as proposed by Kurzak et al. [14] as well as LAPACK [15]. The code divides the matrix into tiles and each process allocates space for the needed tiles. The computation of the tiles and their dependencies is represented with a Directed Acyclic Graph (DAG) of task dependencies.

The algorithm uses four basic operations defined on blocks: DSYRK, DPOTRF, DGEMM and DTRSM. Each function represents a task computing a tile; a more detailed description is provided by Kurzak et al. [14]. Figure 2a shows a simple example of a task graph generated by a computation using 25 tiles.

The simple statically pipelined schedule [14], [16] provides good load balancing and locality if used with dense matrix operations but it targets shared memory architectures. To support distributed memory, we port the original algorithm to MPI Message Passing, MPI One Sided, and to our MPI Notified Access. Our goal is not to provide the best implementation of the Cholesky factorization targeting distributed memory but to compare how different synchronization systems perform with complex and realistic data dependency patterns.

Our implementation broadcasts tiles as needed after they are produced by a process. Data is broadcasted along a binary tree overlay process topology. As soon as a node receives an update, it forwards the update to its children. Due to the complex dependency graph, the asynchronous progression, and the broadcast protocol, nodes generally cannot know what update they receive next. In the MPI message passing version, the indices of the transferred tile are sent coded in the tag. We use a combination of `MPI_Probe` and `MPI_Recv` to determine and post the right address based on the tag of the incoming message.

Since the sender knows the address of the tile it updated, the One Sided implementation seems simple. However, the complexity lies in notifying the target process which tile was updated. In traditional One Sided models, one has to communicate the tile index explicitly to the target. We do this with a ring buffer and atomic compare and swap after the data has been committed remotely. The target processes incoming tiles by polling the incoming tile buffer for new tile addresses. We show an excerpt in the following:

```
MPI_Put(&(tiles_array[x][y]), tile_size, MPI_DOUBLE, target, index,
      tile_size, MPI_DOUBLE, win);
MPI_Fetch_and_op(&one, &dest, MPI_INT, target, 0, MPI_SUM, notifWin);
MPI_Win_flush(target,win);
MPI_Put(&tile_coord, 1, MPI_INT, target, dest, 1, MPI_INT, notifWin);
```

With Notified Access, the notification reporting which buffer completed can be encoded in the source/tag matching. Thus, the source simply issues an `MPI_Put_notify` with a specially crafted tag. The target instantiates a notification request with any source, any tag, and `expected_count`=1. After starting this request, the target uses `MPI_Wait` to receive a notification and reads the correct tag from the returned `MPI_Status`.

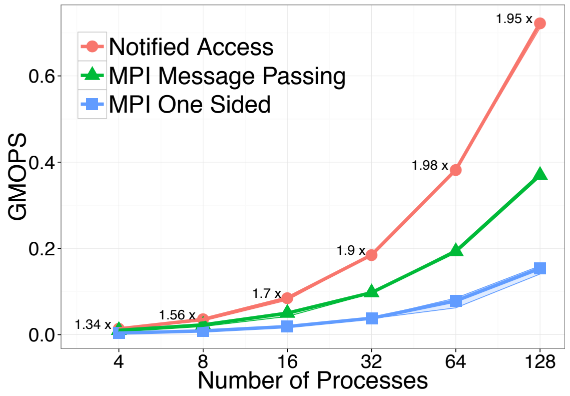


Figure 5: Cholesky factorization performance in a weak scaling experiment with constant transfer size of 8 KBytes

Figure 5 shows the average performance of each version for a weak scaling experiment with a tile matrix size of 32×32 doubles at each process. We repeated each run 10 times and plot the 99% confidence interval as shade. This Cholesky configuration represents an extreme case of a very small computation per process. We use it to demonstrate the small-message efficiency of our implementation of Notified Access for a practically-relevant communication pattern.

VII. Related Work

Notified Access is a synchronization mechanism that abstractly combines benefits of the message passing and RMA programming models. We define it as an abstract concept and demonstrate a specific realization in MPI.

Previous works used one of two schemes for notifications: *counting* or *overwriting* notification identifiers at the target. Our Notified Access design adds a third option to enqueue the notifications at the target to an ordered (matching) list. Counting identifiers accumulate the number of arrived messages or bytes into an integer value while overwriting identifiers can transport an integer value from the source but act as atomic registers at the target. In general, counting interfaces are more scalable than overwriting interfaces because they allow the interface to accumulate message statistics. Overwriting interfaces on the other hand allow to communicate notification values (e.g., tags) from the source but may require more storage at the destination (offer one slot per expected notification). The main benefit of our queuing mechanism is that it combines both as it offers a value (tag) and preserves the arrival order of notifications and the sources do not need to synchronize notification identifiers at the target.

We now discuss and categorize related work chronologically. Split-C [17] provides a counting identifier with its signaling store operation that counts the number of bytes received. The target of signaling stores can wait for a specific number of bytes using the store sync operation. However, this mechanism does not allow to match notifications, characteristic that enables efficient dataflow implementations.

LAPI [18] offers a counting mechanism at the target which allows to count message completions at the target. In ARMCI [19] an undocumented mechanism allows to notify remote processes with synchronization operations. In this communication scheme the origin of a remote write waits for the completion of the write and then sends the notifying message with which the target process can synchronize. This scheme in other word delays the notification to the target process for a round trip time of the network.

The need of providing efficient point-to-point synchronization mechanism was addressed by Bonachea et al. [20] as well. In this work data transfer and synchronization is coupled through the use of signaling puts that are able to synchronize origin and target of a remote access by using counting semaphores. An analogous approach is applied on an extension for SHMEM [21] which utilizes counting puts, that increment remote counters. Schneidenbach et al. [22] propose an overwriting notification mechanism similar to full/empty bits to post remote buffers with a binary notification per buffer.

GASPI [23], [24] provides a flexible overwriting interface with explicit support for multithreaded receiving of notifications. The specification provides ordering guarantees with respect to notifications and related data, but not across multiple notifications. Notified access maintains the arrival order of notifications in the queue and allows to implement serializable semantics on in-order networks. This seems challenging to implement efficiently with GASPI.

All discussed interfaces rely on setting up remote synchronization objects statically before the operation starts. This limits the flexibility of the operation. Our request-based implementation enables flexible matching at the target with only two additional cache misses.

Hori et al. [25] propose a protocol where accesses to specified memory regions trigger notifications automatically. This data-centric approach seems very promising but cannot be supported on most of today's RDMA networks.

Several applications [26], [27] employ custom notification solutions to improve parallel efficiency. Notified access provides a common ground and flexible interface for portable parallel RMA applications.

VIII. Discussion

Notified Access maintains the global memory view and ownership management of RMA and combines it with efficient matched notifications of message passing at lowest overheads. This allows maximum utilization of RDMA hardware and enables producer-consumer schemes. We will now briefly discuss several questions that may arise.

Can network reliability be an issue for Notified Access? While notified writes can be implemented easily in different kinds of networks, notified reads require a more detailed analysis. In Notified Access, the reception of a notification bound to a remote read at the target process, means that a specific buffer can be reused again. Thus, in order to avoid data loss, the notification can only be triggered when can be ensured that the data will arrive at the requesting process. This can be guaranteed in two ways: (1) if the network is fully end-to-end reliable then the notification happens right after the data has been read or (2) if the network is unreliable (may require retransmissions) then the notification can only appear after the data arrived at the process that issued the get. Motivated by the event notification semantics of low-level interfaces such as uGNI or Portals 4, we assume the former case. If the network is not reliable then notified read semantics can be provided using a slightly more complex protocol incurring another network round-trip. This can be implemented with modern technologies such as the Portals 4 [9] by delegating the completion-check and the transmission of an additional synchronization message directly to the NIC, using minimal overhead without involving the CPU.

Can hardware completion counters be utilized? Some networks, e.g., Blue Gene/Q support completion counters where the network interface increments a counter after an access is performed. The current active access design could utilize this functionality with a small extension. If nondeterministic matches are used then the target could contact the source during notification init and set up a static counter for the request. Test and wait would then simply check this counter at lowest overheads.

Does matching at the target inhibit full hardware offload? While message transmission is 100% hardware offloaded, our implementation performs matching using the target CPU. While this is an improvement over message passing where the transfer is never 100% offloaded, the CPU overhead may be high. Yet, today's CPUs are very efficient in the necessary list traversals for matching. In addition, the CPU often waits for notifications and matching messages which causes no additional overhead (cf. helper locks [28]). Thus, we believe that the current solution is most efficient and it can also utilize hardware matching if available (e.g., Portals 4).

Will this be in MPI-4? The MPI Forum recognizes the need for notifications and a proposal for Notified Access looks promising.

What does a network interface require to enable notified access? To ideally support our design, a network interface would enable to add immediate values which should be able to encode a pointer on a machine to a reliable remote queue for each remote access (put and get). Cray's uGNI is close to this specification (yet, it only offers 32 bit values, for example).

IX. Conclusions

In this paper we present a new matched synchronization paradigm for RMA programming called Notified Access. It is motivated by the inability of most RMA synchronizations to exploit the full network performance for producer-consumer communications. Notified Access leverages modern interconnect features to create a hardware-offloaded lightweight synchronization mechanism that is particularly efficient for asynchronous small latency-limited messaging.

Notified Access extends previous notification mechanisms with matching queue semantics. These new semantics addresses scalability problems by moving the notification placement decision from the origin to the target process while providing intuitive properties such as maintaining the arrival order in the queue. This allows to offer strong consistency of accesses and notifications on in-order networks.

We extend the MPI interface to include matched notified access, an interface providing convenient semantics to differentiate accesses by source and a user-defined tag at the target. We show how our interface can be implemented with minimal overheads and we demonstrate its efficiency for various use-cases. Our experiments also identify three parallel patterns that naturally fit many applications. Just to name one example, our fine-grained dataflow implementation demonstrates speedups up to 2x over Message Passing for a small Cholesky factorization. We expect that Notified Access will be an important primitive for exploiting future large-scale networks towards exascale.

Acknowledgments

We thank Hatem Ltaief (Kaust) for providing the Cholesky example. We thank the GASPI team for inspiring discussions about RMA interfaces and Christian Simmendinger for numerous clarifications about the GASPI specification. We thank James Dinan (Intel), Jeff Hammond (Intel), Kathy Yelick (LBNL), Edgar Solomonik, Timo Schneider, and Salvatore Di Girolamo for helpful discussions, Larry Kaplan (Cray) for help with uGNI, and the Swiss National Supercomputing Centre (CSCS) for access to Piz Daint.

References

[1] W. Huang, G. Santhanaraman, H. Jin, Q. Gao, and D. Panda, "Design of high performance MVAPICH2: MPI2 over infiniband." in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1, May 2006, pp. 43–48.

[2] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap." in *Proceedings of the International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE Computer Society, 2006, pp. 1–10.

[3] H. Shan, B. Austin, N. Wright, E. Strohmaier, J. Shalf, and K. Yelick, "Accelerating applications at scale using one-sided communication." in *Proceedings of the Conference on Partitioned Global Address Space Programming Models (PGAS'12)*, 2012.

[4] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling highly-scalable remote memory access programming with MPI-3 one sided." in *IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC13)*. ACM, Nov. 2013, pp. 53:1–53:12.

[5] T. Mattson, R. van der Wijngaart, "Parallel research kernels." available at: https://github.com/ParRes/Kernels (Aug. 2014).

[6] MPI Forum, *MPI: A Message-Passing Interface Standard. Version 3.0*, September 21th 2012, available at: http://www.mpi-forum.org (Sep. 2012).

[7] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, and A. B. Maccabe, "High performance RDMA protocols in HPC." in *Proceedings of the 13th European PVM/MPI User's Group Conference*, ser. EuroPVM/MPI'06, 2006, pp. 76–85.

[8] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing - to thread or not to thread?" in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.

[9] B. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, W. K., K. Underwood, R. Reisen, A. Maccabe, and T. Hudson, *The Portals 4.0 network programming interface*, Sandia National Laboratories, November 2012, Technical Report SAND2012-10087.

[10] M. ten Bruggencate and D. Roweth, *DMAPP - An API for One-sided Program Models on Baker Systems.*, Cray User Group (CUG), 2010.

[11] M. Woodacre, D. Robb, D. Roe, and K. Feind, "The SGI Altix TM 3000 global shared-memory architecture." 2003.

[12] Cray Inc., *Using the GNI and DMAPP APIs. Ver. S-2446-52.*, March 2014, available at: http://docs.cray.com/.

[13] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the LogP model: One step closer towards a realistic model for parallel computation." in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95. New York, USA: ACM, 1995, pp. 95–105.

[14] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 1, pp. 15–44, Jan. 2010.

[15] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "LAPACK: A portable linear algebra library for high-performance computers." in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '90. IEEE Computer Society Press, 1990, pp. 2–11.

[16] J. Kurzak, A. Buttari, and J. Dongarra, "Solving systems of linear equations on the CELL processor using Cholesky factorization." *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 9, pp. 1175–1186, 2008.

[17] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. Von Eicken, and K. Yelick, "Parallel programming in Split-C." in *Supercomputing'93*. IEEE, 1993, pp. 262–273.

[18] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, "Performance and experience with LAPI — a new high-performance communication library for the IBM RS/6000 SP." in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998*. IEEE, 1998, pp. 260–266.

[19] J. Nieplocha and B. Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems." in *Parallel and Distributed Processing*. Springer, 1999, pp. 533–546.

[20] D. Bonachea, R. Nishtala, P. Hargrove, and K. Yelick, "Efficient point-to-point synchronization in UPC." in *Proceedings of 2nd Conf. on Partitioned Global Address Space Programming Models*, Oct. 2006.

[21] J. Dinan, C. Cole, G. Jost, S. Smith, K. D. Underwood, and R. W. Wisniewski, "Reducing synchronization overhead through bundled communication." in *First Workshop, OpenSHMEM 2014*, Annapolis, MD, 2014, pp. 163–177.

[22] L. Schneidenbach, D. Böhme, and B. Schnor, "Performance issues of synchronisation in the MPI-2 one-sided communication API." in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 177–184.

[23] D. Grünewald and C. Simmendinger, "The GASPI API specification and its implementation GPI 2.0." in *7th International Conference on PGAS Programming Models*, vol. 243, 2013.

[24] C. Simmendinger, M. Rahn, and D. Gruenewald, "The GASPI API: A failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures." in *Sustained Simulation Performance 2014*. Springer, 2015, pp. 17–32.

[25] A. Hori, J. Lee, and M. Sato, "Audit: A new synchronization api for the GET/PUT protocol." *J. Parallel Distrib. Comput.*, vol. 72, no. 11, pp. 1464–1470, Nov. 2012.

[26] M. Krishnan, R. Lewis, and A. Vishnu, "Scaling linear algebra kernels using remote memory access." in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, Sept 2010, pp. 369–376.

[27] T. P. Straatsma and D. G. Chavarra-Miranda, "On eliminating synchronous communication in molecular simulations to improve scalability." *Computer Physics Communications*, vol. 184, no. 12, pp. 2634–2640, 2013.

[28] K. Agrawal, C. E. Leiserson, and J. Sukha, "Helper locks for fork-join parallel programming." in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 245–256.