SAND2013-2920C

Protocols for Fully Offloaded Collective Operations on Accelerated Network Adapters

Timo Schneider, Torsten Hoefler
ETH Zurich
Dept. of Computer Science
Universitätstr. 6
8092 Zurich, Switzerland
Email: {timos,htor}@inf.ethz.ch

Ryan E. Grant, Brian Barrett, Ron Brightwell
Sandia National Laboratories*
P.O. Box 5800, MS-1110
Albuquerque, NM 87185-1110
Email: {regrant, bwbarre, rbbrigh}@sandia.gov

***Abstract*—With each successive generation, network adapters for high-performance networks are becoming more powerful and feature rich. High-performance NICs can now provide support for performing complex group communication operations on the NIC without any host CPU involvement. Several "offloading interfaces" have been designed with the collective communications goal being the *complete* offloading of *arbitrary* communication patterns. In this work, we analyze the offloading model offered in the Portals 4 specification in detail. We perform a theoretical analysis based on abstract communication graphs and show several protocols for implementing offloaded communication schedules. Based on our analysis, we propose and implement an extension to the portals 4 specification that enables offloading *any* communication pattern completely to the NIC. Our measurements with several advanced communication algorithms confirm that the enhancements provide good overlap and asynchronous progress in practical settings. Altogether, we demonstrate a complete and simple scheme for implementing arbitrary offloaded communication algorithms and hardware. Our protocols can act as a blueprint for the development of communication hardware as well as middleware while optimizing the whole communication stack.**

I. Motivation

Moore's law is still going strong despite the end of frequency and Dennard scaling. CPU and chip vendors have managed to maintain Moore's law by going *broad*, i.e., duplicate functional units (e.g., cores or vector units) and/or add new functionality to chips. Thus, several microprocessor vendors began extending core functionalities of chips. For example, functionalities that were traditionally in a north bridge, such as a memory controller, are now commonly included in main CPUs. Similarly, networking chips have become more powerful and are to be integrated into next-generation CPUs.

The growing number of cores per network endpoint increases the requirements for the network and memory interfaces. Modern multi-core CPUs already scale the number of memory controllers with the cores, similarly, network interfaces may follow. High-performance networks provide much more complex functionality than memory controllers. Thus, it seems reasonable to devote some silicon to performing advanced functions.

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

The most important parameters of today's networks are latency, bandwidth, and message-rate. Therefore, modern networks are highly tuned to provide high performance for these metrics. However, many algorithms from scientific computing and other fields, such as databases, operating systems, and financial computations, use advanced communication algorithms over sets of processes. These are often called "collective communications" in high-performance computing (HPC) and they are important to many types of applications. Their increased complexity over standard point-to-point communications and participation of multiple processes make them important to overall communication performance, as well as a prime target for efforts to enhance network performance.

Several network interfaces such as Quadrics, Myrinet, and Mellanox ConnectX2 include hardware that can support the execution of collective operations directly on the network interface without involving the (relatively distant) CPU. This also allows the CPU to perform other tasks asynchronously instead of serving network interrupts. Current networks are limited to specific hardware and MPI operations. A more flexible solution that can support arbitrary communication topologies would lessen these existing limitations.

In this work, we aim to generalize the collective communication problem to describe interfaces and protocols for *accelerated* network offloading. We generalize collective algorithms to arbitrarily complex communication topologies or *overlay networks*. We use several existing designs to derive protocols and an interface extension to an existing "network offload" mechanism to enable *full offload (acceleration)* of arbitrary communication algorithms.

Fully offloading communication algorithms provides several benefits: (1) it directly reduces latency by avoiding interaction with the main CPU and thus speeds up applications; (2) it reduces the power consumption of a system by saving data transfers between different domains (NIC, CPU); (3) it enables the CPU to perform computations without interruption to serve communications, and (4) it reduces the influence of small performance variations ("System noise" [1]) to large-scale applications.

Accelerated network interfaces are relevant for off-chip as well as on-chip communications. Off-chip communications often suffer from higher latencies that can be reduced by handling communication algorithms in the network interface.

Such additional latencies can be up to 500 ns while network transfers only take 800 ns. Indeed, Underwood et al. demonstrated a 30% latency improvement due to communication offload [2]. It's less obvious for on-chip networks since they are not suffering from the latency issue and the data is always close, however, very simple cores can suffer significantly from pipeline stalls and network interrupts. Thus, a simple communication offload engine can be very useful in such a situation as well.

As discussed before, current interfaces are either limited to certain hardware configurations or to communication algorithms. The most flexible interface is the Portals 4 specification [3] that offers so called "triggered operations" which can be used to perform several actions when messages arrive. This allows one to implement a basic set of algorithms [1], [2]. However, the current interface is limited and does not allow full offload. Thus, we discuss the limits of this approach, propose several protocols to enable as much offloading as possible, and discuss a simple extension that will enable full offloading capabilities.

To summarize, the key contributions of our work are:

- We discuss several practical issues such as schedule reuse and cross-matching and provide protocols to enable an implementation of our approach.
- We demonstrate our implementation using a Portals 4 reference implementation with a rich set of collective algorithms that can all be run without intervention of the CPU.
- We show how to design an interface that enables *full* offload for *arbitrary* communication schedules enabling the implementation of *accelerated overlay networks with arbitrary topologies*

Our discussions provide new insights into the requirements of communication algorithms and our interfaces (extended Portals 4) and protocols can directly be used by hardware designers to develop a generic network offload infrastructure.

We first start with a discussion of necessary semantics to implement communication algorithms followed by a discussion of Portals 4. We then demonstrate protocols to implement fully offloaded communication algorithms over Portals 4 and we raise several issues. We then provide a proposed extension to Portals 4 to better support offloading arbitrary collective communication schedules. In our experimental evaluation, we demonstrate the usefulness of our approach in practice and conclude with a discussion and outlook.

II. Communication schedules as dependency graphs

Message Passing Interface (MPI) [4] libraries implement collective operations as communication algorithms and are thus a good example for our discussion. In such libraries, e.g., MPICH [5] or Open MPI [6], collective operations are commonly built on top of point-to-point communications. Each collective implementation is a piece of (most often C) code that calls basic point-to-point or RDMA communication primitives. An example of such an implementation is given in Listing 1. In this simple barrier implementation each non-root node signals its arrival at the barrier by sending a zero-length message to the root and then waits for another zero length message from the root which signals the end of the barrier. The wait step is implemented using a blocking receive operation.

The root process uses non-blocking communication since it sends and receives multiple messages. Non-blocking communication primitives allow the parallel processing of messages in this case. The root explicitly waits for communication steps to finish using the wait_all() function.

```
/*All non-root send & receive zero-length message.*/
if (rank > 0) {
  send(NULL, 0, MPI_BYTE, 0, i, ...);
  recv(NULL, 0, MPI_BYTE, 0, MPI_ANY_SOURCE, ... );
}
/*The root collects and broadcasts the messages.*/
else {
  for (i = 1; i < size; ++i) {
    irecv(NULL,0,MPI_BYTE,MPI_ANY_SOURCE,...,&(reqs[i]));
  }
  wait_all(size-1, reqs+1, MPI_STATUSES_IGNORE);
  for (i = 1; i < size; ++i) {
    err = isend(NULL, 0, MPI_BYTE, i,...,&(reqs[i]));
  }
  wait_all(size-1, reqs+1, MPI_STATUSES_IGNORE);
}
```

Listing 1. Open MPI Barrier implementation for small communicators

This way of implementing collective operations is well suited for a host CPU centric message passing framework. However, it is rather difficult to use such an implementation for offloading the collective execution to a co-processor. As the collective is implemented in C, the co-processor itself would have to be able to execute (compiled) C code. The code above is also hard to understand because dependencies between send and receive operations are expressed in two different ways. In the non-root case the receive cannot start before the blocking send is completed. This is not explicitly stated, it is a result of the control flow in the program. In the root case the dependencies are made explicit using wait_all().

This makes it hard to reason about properties of collectives implemented in such a way. It is not clear if the implicit dependency in the non-root case is a side-effect of the control flow or necessary for the communication.

Hoefler et al. [7] describe the idea of expressing collective operations as a dependency graph between simple networking primitives, such as send and receive in a system called cDAG. They also included the possibility to specify transformations on process-local data.

A. Semantics of cDAG

In the following we will describe the semantics of cDAG, a communication framework based on those ideas.

Communication patterns are expressed as a graph $G = (V, E)$. The vertices V of this graph are send and receive operations as well as arithmetic operations on local data, the edges of this graph are dependencies. An edge $e = (u, v)$ between two operations, u and v, means that the operation v must not *start* before the operation u has *completed*.

A send operation s is defined as a tuple including buffer, length, source, destination, and matching tag:

$(s_{buf}, s_{len}, s_{src}, s_{dst}, s_{tag})$, the definition of a receive is likewise $(r_{buf}, r_{len}, r_{src}, r_{dst}, r_{tag})$. The complete communication algorithm is finished as soon as all vertices are *completed*. We define the completion of sends and receives as follows:

A send is completed as soon as it is guaranteed that modifications of the memory range $mem[s_{buf}, s_{buf} + s_{len}]$ at rank s_{src} do not change the contents of the message delivered to the remote rank. Likewise, a receive is completed as soon as the memory range $mem[r_{buf}, r_{buf} + r_{len}]$ contains the value delivered by the matching send. That is, any load from that address range (in a send or by the host CPU) will result in reading the new values.

Messages can have tags to identify different channels. Receives only "match" with sends that have identical tags as specified in the receive (a special wildcard tag is available). In the following we will describe the matching semantics of cDAG: If a send operation s is started (which means that the cDAG schedule execution was started and for all edges $(x, s) \in E$, x has completed), it is added to the set of outstanding sends. Similar, when a receive is started, it is added to the set of outstanding receives. The (partial) order in which elements are added to the outstanding sends and receives set is defined by the happens-before relation introduced by the dependencies, as well as matches. If the outstanding sends set contains an element s and the outstanding receives set contains an element r with $s_{dst} = r_{dst} \wedge s_{src} = r_{src} \wedge s_{tag} = r_{tag} \wedge s_{len} = r_{len}$ those two elements *match*, both are removed from the outstanding sends and outstanding receives set, before other vertices are added. When two operations s and r match we show this, where needed, by drawing a "matching edge" (dashed arrows in Figure 1). Edges introduced by deterministic matches imply a similar happens-before relationship than dependency edges: if there exists a matching edge (s, r) the receive operation r can not finish before s is started (whereas, if it were a dependency-edge, r could not start before s was finished).

Since the happens-before relation introduced by dependencies is only partial, it is possible to construct cDAG graphs which contain multiple possibilities to match sends to receives, such as shown in Figure 1. In such a case it is undefined which send will be matched to which receive.

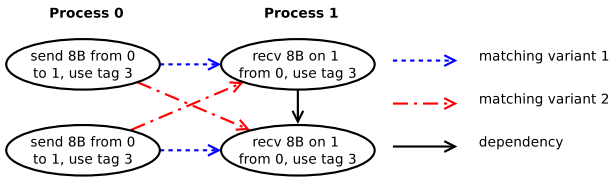

Fig. 1. Non-deterministic matching example: The cDAG semantics do not define which send matches which receive

There are no dependencies (neither a directed edge, nor a path) between the two sends, therefore the scheduler is free to start them in any order and thus allow both matchings shown. If a cDAG schedule contains non-determinism, the user has to ensure that all possible matchings are correct.

To perform collective communication with cDAG, a cDAG graph has to be assembled first, by adding operations and dependencies between them. Graph assembly is process-local. When the cDAG graph is complete it has to be compiled in a collective call. This enables the runtime to perform optimizations on the graph or change its internal representation. A compiled graph can then be executed as a non-blocking collective operation. To test for completion, the user can use a non-blocking test-function or a blocking wait-function.

With those semantics all communication patterns that can be expressed in the message passing paradigm can be captured.

B. Collective Operations in cDAG

The cDAG abstraction allows the implementation of collective operations in a convenient manner. We implemented several variants of the collectives defined by the MPI standard in the libNBC library [8]. Originally libNBC implemented collective operations in a round-based fashion, all communication operation in a round had to be finished before the next round could start. We changed its internal scheduler to utilize cDAG graphs to enable a higher degree of flexibility in expressing dependencies between communication operations. We are now able to link against different cDAG backends, i.e., for MPI, Cray DMAPP [9], or Portals 4, as presented in this paper. Table I shows all of the variants of collectives that have been implemented.

The variants listed in Table I have not been implemented in code, unlike what a typical MPI implementation would do. Instead we leveraged the graph-based nature of cDAG to define a dataflow graph (which defines which data item has to be communicated from where to where) and a topology graph (which defines which communication topology, i.e., a binomial tree or a ring) the dataflow graph should be mapped onto. The created cDAG schedule then combines both graphs. This allows us to provide many variants for each collective communication function; for example the "Tree" variant is able to be used with k-ary as well as with k-nomial trees, for different values of k, by changing a single parameter in a function call. This vast set of varieties could be used for auto-tuning collectives during run-time in the future.

III. A Case Study: Portals 4

We now discuss how to use triggered operations to implement full offloading of arbitrary communication patterns that have been specified in cDAG. These operations start when a counter reaches a predefined threshold. The supported operations are: PtlPut, PtlGet, PtlAtomic, PtlFetchAtomic, PtlSwap, PtlCTInc, and PtlCTSet.

Portals supports both matching and non-matching communication mechanisms. The matching interface offers much greater flexibility in target side message handling, at the cost of message reduced processing rate. In this text we will only consider the matching interface because matching is needed to offload complex communication schedules. Without it one can not differentiate between different messages without host CPU involvement. Before data can be sent or received over Portals, the network interface has to be initialized, but we will not concern ourselves with the initialization process here.

If we want to send data from one node to another, we have to set up a memory-descriptor data structure. This contains

Collective	Implemented Variants
MPI_Broadcast	Pipelined k-ary and k-nomial Tree, Pipelined Linear, round-optimal
MPI_Scatter	Pipelined k-ary and k-nomial Tree, Pipelined Linear
MPI_Gather	Pipelined k-ary and k-nomial Tree, Pipelined Linear
MPI_Allgather	Ring, Butterfly, Pairwise Exchange, Dissemination, Broadcast after Gather
MPI_Barrier	Butterfly, Pairwise Exchange, Dissemination, Broadcast after Reduce
MPI_Alltoall	Pairwise Exchange
MPI_Reduce	Pipelined k-ary and k-nomial Tree, Pipelined Linear
MPI_Reduce_scatter	Pairwise Exchange, Scatter after Reduce
MPI_Allreduce	Butterfly, Dissemination, Broadcast after Reduce
MPI_Scan	Pipelined Linear
MPI_Exscan	Pipelined Linear

TABLE I. COLLECTIVES IMPLEMENTED IN LIBNBC USING THE CDAG ABSTRACTION

information about the start and length of the described buffer. We can also choose to let Portals count events associated with this memory region, for example how many times this buffer has been sent. If we want to do such counting, the counter handle is also included in the memory descriptor data structure. The counter itself is set up with PtlCTAlloc. After the memory descriptor is set up, it is registered with Portals with the PtlMDBind. A more detailed code snippet is shown in Listing 2.

```
ptl_md_t send_md;
ptl_handle_me_t send_md_handle;

send_md.start = &one;
send_md.length = sizeof(uint64_t);
send_md.options = PTL_MD_EVENT_CT_SEND;
send_md.eq_handle = PTL_EQ_NONE;
PtlCTAlloc(ni_logical, &send_md.ct_handle);
PtlMDBind(ni_logical, &send_md, &send_md_handle);

PtlPut(send_md_handle, 0, sizeof(uint64_t), PTL_NO_ACK_REQ,
     peer, logical_pt_index, 1, 0, NULL, 0);
PtlCTWait(recv_value_me.ct_handle, 1, &ce);
```

Listing 2. Sending data with Portals

After we set up the memory descriptor we can pass it to a remote process using PtlPut. The last line in this Listing shows one way of utilizing the counter that we set up: PtlCTWait will block until the specified counter reaches a given value. But simply waiting for the counter to reach a specified value does not help us to offload a more complicated communication pattern. Therefore Portals also offers a way to trigger new operations, based on counting events.

To receive data we have to set up a match list entry. The match list entry contains information on the source of the incoming message and how to handle it. We can also specify from which rank we expect a message and from which user. Further, we can specify 64 match bits, which are masked by the same amount of ignore bits. In the previous example we set the value of the match bits to 1. We can receive this message as follows:

```
recv_value_me.start = &will_be_one;
recv_value_me.length = sizeof(uint64_t);
PtlCTAlloc(ni_logical, &recv_value_me.ct_handle);
recv_value_me.uid = PTL_UID_ANY;
recv_value_me.match_id.rank = PTL_RANK_ANY;
recv_value_me.match_bits = 1;
recv_value_me.ignore_bits = ~1;
recv_value_me.options = PTL_ME_OP_PUT | PTL_ME_EVENT_CT_COMM
      | PTL_ME_EVENT_COMM_DISABLE;
PtlMEAppend(ni_logical, 0, &recv_value_me, PTL_PRIORITY_LIST
     , NULL, &recv_value_me_handle);
```

Listing 3. Receiving data with Portals

First, we specify where the message should be stored and how long it is. Then we allocate another counter, which we will use to count the Put commands that wrote to that buffer. We will accept messages from any rank and any user. Furthermore one can specify a 64 bit wide "tag" the message must have to match to this match list entry with the match_bits field. The ignore_bits field specifies which bit of the 64 match bits should be ignored for matching. After the match entry is complete we pass it to Portals with the PtlMEAppend function. This function inserts the match list entry at the end of the match list. If a message does not match any match list entry, it is dropped by the receiver.

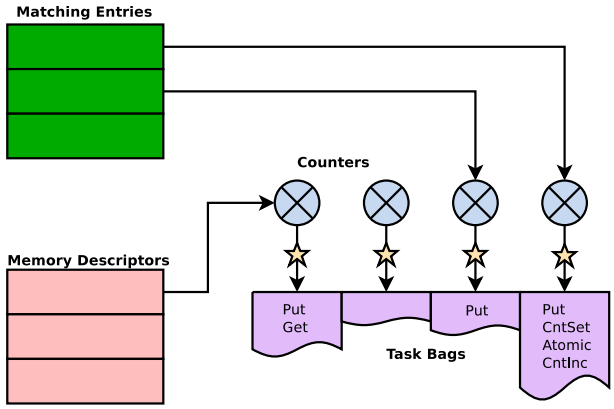

Fig. 2. Simplified overview of the trigger mechanism in Portals: A match list entry and memory descriptors can increment a counter upon certain events. Triggered operations can be set up to be executed when a counter reaches a predefined threshold. In our drawings we use stars to symbolize the trigger event caused by reaching the threshold value.

In Figure 2 we show an overview of the components involved in Portals triggered operations. We have to create memory descriptors and match list entries to send and receive data. Memory descriptors can be associated with a counter which can count the number of sends (put), reply events (get), acknowledgments, or the number of send/received bytes. Match list entries can also be associated with a counter and they are capable of counting the number of matched put, get and atomic operations (or the number of affected bytes). The counters are associated with triggered operations (put, get, atomic, counter-increment, and counter-set) which are executed when the counter reaches a certain threshold. The thresholds can be set for each triggered operation independently. Note that this does not reflect all data structures and constructs actually used by Portals, it shows only those relevant to triggered operations and the matching interface.

In both examples we did not use the offload features that Portals offers. With Portals we can make the *hardware* react to incoming messages *without* involving the host CPU. The way this is done is shown in Figure 3.

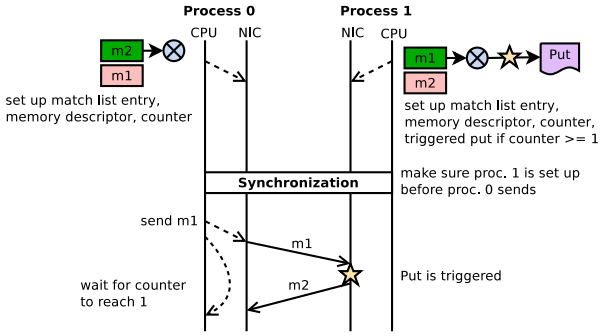

Fig. 3. Triggered response to a message: After the relevant data structures are set up (c.f., Fig 2) the m1 matches a match list entry at the receiver, this match increments a counter, which triggers the execution of a PtlTriggeredPut. The numbers identify the different trigger events.

As shown in the previous code listings we have to set up the memory descriptors for sending and the match entries for receiving on each process. For both processes we add a counter to the matching entry to count the number of matched messages. On process 1 we use this counter in a PtlTriggeredPut operation, which executes a PtlPut as soon as the specified counter reaches a certain threshold. After these preparations, we need to synchronize the processes to ensure that process 0 will not send m1 before process 1 has posted the corresponding match entry. After we send m1 from process 0, we can perform a blocking wait on the counter attached to process 0's matching entry with the PtlCTWait function. Note that we do *not* have to give the control to Portals again after the initialization on process 1 to make the ping-pong work. Portals triggered operations provide full asynchronous progress, so we do not need periodic calls to a test-function like in MPI (see [10] on progressing non-offloaded communication) to ensure progress.

Portals does not allow triggered changes to the entries in the match list or in the memory descriptor list. However, it is possible to define whether a matching entry should be persistent or match only once.

IV. Fully Asynchronous Collectives

We now discuss how triggered operations can be used to implement a cDAG interpreter. We will show how we use common protocols that have been used for CPU-driven applications in the context of *full offloading of arbitrary communications* to the network interface.

The point-to-point protocol design is important for cDAG-over-Portals because, unlike MPI, Portals does not buffer messages. On the other hand it does not require an exchange of information (i.e., pre-calculated destination addresses) between sender and receiver before any data can be sent like RDMA, as Portals supports message matching. In this section we explain the options for implementing cDAG semantics over Portals.

Of course we can always emulate buffering semantics by providing shadow buffers (buffers that are allocated by the runtime in the compile phase) on the receiver for each message that we will receive in the execution phase. This approach requires additional memory and messages have to be copied from the shadow buffers to the user buffers at the receiver. The advantage of such an approach is that this protocol does not need any control messages and does not send data needlessly. Therefore such an *eager* protocol should be used for small messages, where the buffering does not cause too much overhead.

If we do not buffer unexpected messages we have to guarantee that the sender does not overwrite the message by receiving into the same buffer again before the message was matched by the receiver. Ensuring this kind of synchronization will always require additional messages. An overview of how point-to-point messaging can be implemented is shown in Figure 4.

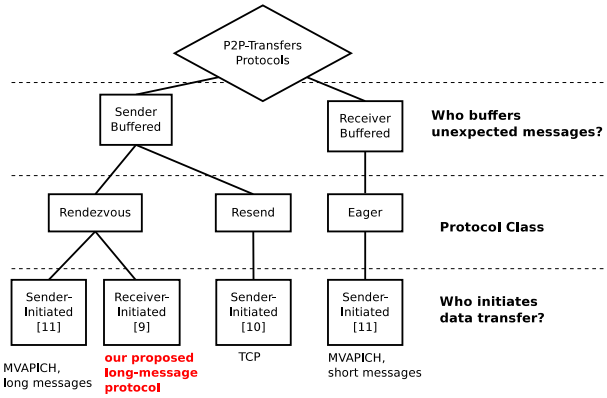

Fig. 4. Hierarchy of point-to-point protocols: The eager protocol ensures reliable delivery by storing all incoming messages at the receiver, dequeuing them when a matching receive is posted. The rendezvous protocol ensures that the receiver already posted the receive. A third option is to resend messages until they are acknowledged by the receiver.

Buffering at the sender with a rendezvous protocol can be done in two ways: Either the sender initiates the message transfer, by sending a message header to the receiver, and the receiver buffers those (small) message fragments and replies when he is ready to receive the message (or used a RDMA Get operation to receive it). We call this sender-initiated transfer. Another approach is for the receiver to signal when he has the receive buffer for a specific message available, upon receiving such a signal the sender replies with the message data. We call this approach receiver-initiated. For messages below a certain threshold length, it is often more efficient to copy them to a temporary buffer at the receiver, until the receive buffer is ready. This avoids synchronization but needs more buffer space. This approach is called "eager protocol". The third approach is to transmit a message and wait for an acknowledgment from the receiver. If this acknowledgment is not received before a timeout, the message is retransmitted. Such a protocol is rarely used in high-performance networks because it wastes bandwidth and increases the latency because a late receiver always has to wait until the next timeout elapses at the sender.

A. Fully offloaded Rendezvous Protocols

By combining/chaining counters we can execute arbitrary sequences of put/get operations. However, we can not influence if/to which match list entry a message matches asynchronously. In [14], Barrett et al. propose a "triggered rendezvous protocol", however, the *decision* if a message is an expected or unexpected message can not be made without host CPU involvement. In Figure 3 in their paper they use a PtlMEAppend call to differentiate between the two, in the text they write "If the message is expected, the first part of the message is delivered directly into the receive buffer, otherwise it is delivered into bounce buffers". This differentiation is done by the aforementioned PtlMEAppend call. Since Portals offers no triggerable functions to modify the match list, it is impossible to implement a sender-initiated rendezvous protocol (where the receiver has to decide if a message goes to the receive- or a bounce-buffer) with Portals if we demand that the state-change of a receive from "not-posted" to "posted" should be done without host involvement.

Without host CPU involvement, the only way we can change the status of a receive from non-posted to posted is by incrementing a counter, this in turn can trigger puts, gets or other counters to be incremented. So a possibility for implementing a truly asynchronous rendezvous protocol, where the state-change of a message is done without host CPU involvement, is to have a separate memory space on each node, which collects the state of all receive operations for which the node is a possible sender. A local state-change from non-posted to posted receive would then trigger a send to this table. Each entry in this table belongs to a separate match list entry, which in turn triggers the send of the message. The problem with this is that it only works with pre-matched messages, because we have to know for each receive which remote send it matches to!

The pre-matching of messages removes all non-determinism from the cDAG schedules and the pre-matching itself costs time. This is acceptable for collectives that are executed repeatedly (with the same buffer arguments, so that the underlying cDAG schedule they are translated to stays the same). We analyzed the overhead of dynamic matching in [15].

Corollary: *We conclude that, to support the offload of arbitrary communication topologies, the accelerator should be able to change the match list, otherwise pre-matching of messages is required.*

B. Offloaded Rendezvous Protocol with pre-matched Messages

In the following we describe how a rendezvous protocol, which does not involve the host CPU, can be expressed using triggered operations if we assume pre-matched messages. The basic idea is to trigger the sending of a ready-to-receive-flag at the receiver, which uses tags to match a specific matching entry at the sender, where it triggers the send of the actual message. Figure 5 shows the flow of messages in this protocol.

Each of the n send/receive operations in the local schedule has a unique id $\in (0...n-1)$. For each of the operations we allocate one byte of local memory. This byte is called the "status-flag" of the corresponding operation. The addresses

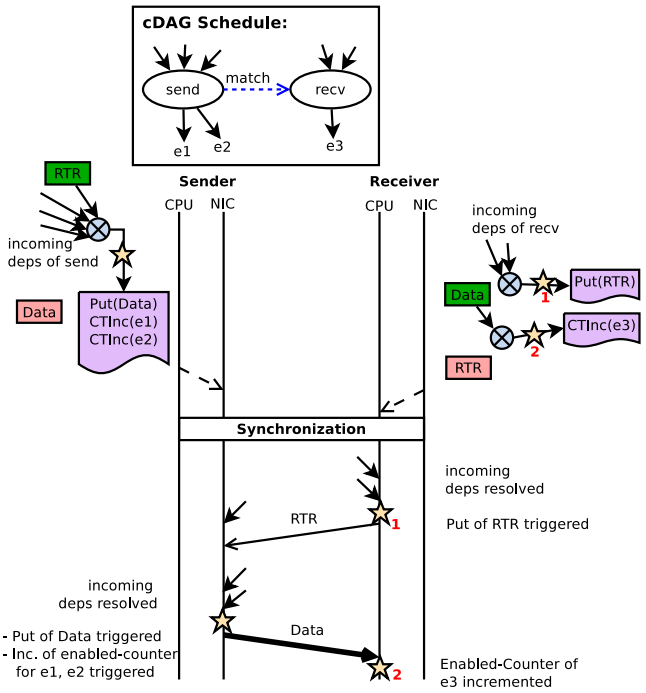

Fig. 5. Rendezvous protocol used to express pre-matched cDAG graphs with Portals triggered operations

for each receive status flag are communicated to the matching send.

For each receive there is an enabled-counter which is incremented once when an operation the receive depends on is finished. For each receive there is also a triggered put operation which is triggered at a threshold of $indeg(id)$, the number of incoming edges (or direct dependencies) of the receive operation in the cDAG graph. The triggered put sends a message to the sender which matches a match list entry which makes sure that the message gets received into the status-flag of the corresponding receive. We use the match bits to differentiate between those control messages and messages carrying user data. The matching match list entry will increment the counter for the send. Each send has a triggered put operation set up, which is triggered at a threshold of $indeg(id)+1$ and will send the user data to the receiver, where it is matched by a match list entry which causes the enabled-counter of each dependent operation of the receive to be incremented by one, since the receive is now finished.

Figure 5 gives an example of how a portion of a cDAG schedule can be translated into Portals triggered operations. Note that this protocol requires synchronization across all participating processes to perform the pre-matching as well as to ensure that all match list entries are posted before any process sends its first message. Since the MPI standard mandates that "all calls [to non-blocking collectives] are local and return immediately, irrespective of the status of other processes" a non-blocking barrier has to be used in order to be MPI compliant. The setup of this non-blocking barrier can be done during the creation of each communicator or group. Upon completion of this non-blocking barrier, the initial sends and receives (which do not have incoming dependencies

in the initial cDAG schedule) are triggered. Note that if an implementation wants to allow n outstanding non-blocking collectives it has to prepare n such non-blocking barriers.

For all protocols described in this work, the tag space has to be used to ensure three different things:

- To discriminate between the different types of control messages (ready-to-send, ready-to-receive, actual data messages, etc.). A small number of bits is sufficient for this task.
- To discriminate between simultaneously outstanding schedules. Since all schedules are started collectively we can maintain local free-lists of schedule-tags.
- To support tags in cDAG operations itself. If a cDAG operation uses ANY_TAG in a receive we mask out this part of the match bits.

C. Offloaded Rendezvous Protocol without pre-matched Messages

For a fully offloaded rendezvous protocol, which does not require pre-matched messages we need some way of manipulating the match list with triggered operations. Probably the simplest operation of such a kind is PtlTriggeredMEAppend together with a triggered version of PtlMEUnlink. This is not yet a part of the released Portals API, but we implemented it as a possible extension to make it more versatile.

We use this proposed functionality in the rendezvous protocol shown in Figure 6.

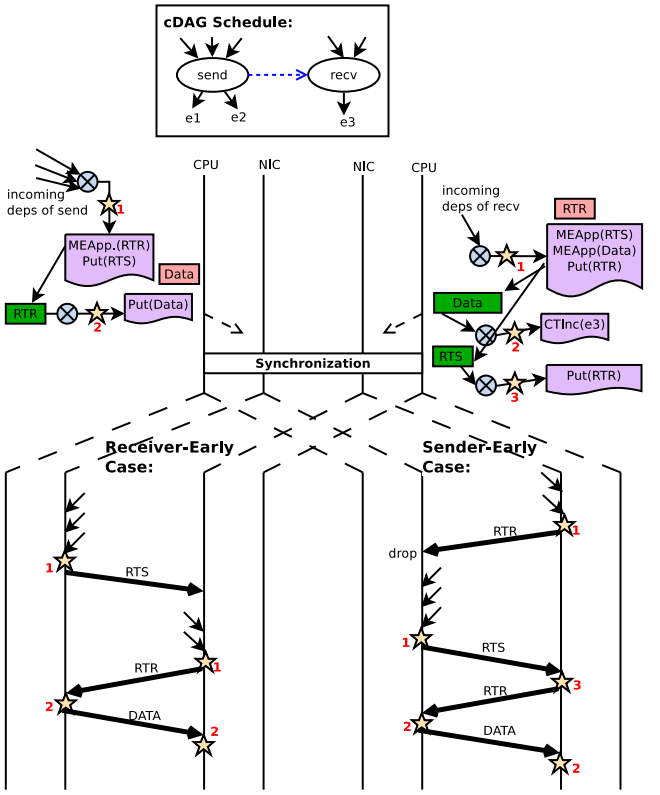

Fig. 6. Rendezvous protocol used to translate cDAG to Portals triggered operations without pre-matching of messages.

When a send is activated (all nodes at the tails of incoming edges in the communication DAG are finished) a match list entry for a "ready-to-receive" (RTR) message is appended to the match list (via PtlTriggeredMEAppend). The message types are discriminated by using two bits of the (64 bit) tag space. The trigger that "enables" the send operation also triggers a put operation of a "ready-to-send" (RTS) message to the peer. The match list entry for the RTR message triggers a put of the actual data. The fact that the receiver sends the RTR message before the data is transferred ensures that the receiver has posted a match list entry for the data.

On the receiver side we append a match list entry for the RTS together with the match list entry for the actual data we expect in this receive. We count the matches on the RTS match list entry and if we detect a match of an RTS we send a new RTR message, in case the receiver dropped the previous one. Each match list entry is set up to match only once.

D. Offloaded Eager Protocol

For small messages the latency added by the synchronization step of the rendezvous protocol is high compared to the latency of the actual data transfer. Furthermore, if a node only receives a small number of such small messages, the available memory at the node is enough to allow buffering such messages. For this reason most available MPI implementations, such as MPICH and Open MPI, use a threshold value (often around 4 KB message size) to decide which messages should be sent using an eager protocol, and which messages should be sent using a rendezvous protocol. With the semantics provided by Portals 4 triggered operations it is possible to implement a fully offloaded eager protocol.

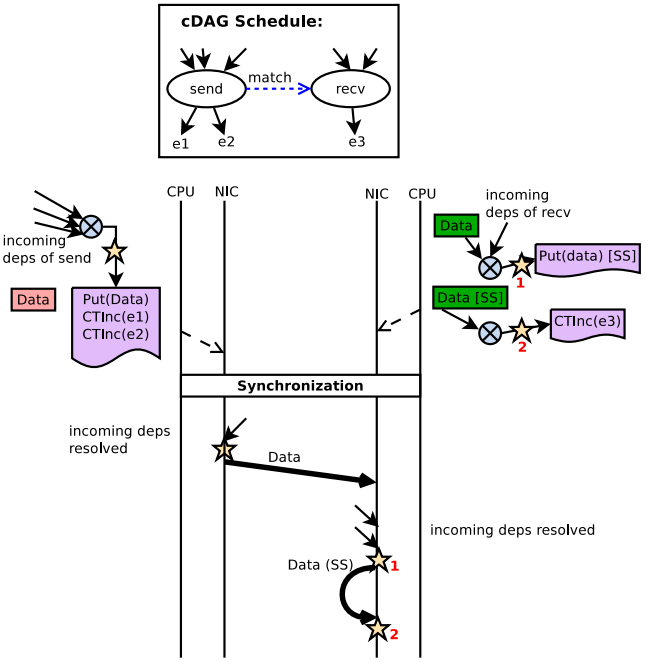

Fig. 7. Eager protocol used to execute pre-matched cDAG schedules using Portals triggered operations

For each small message receive operation r we allocate a shadow buffer of the size $r.size$. For each message that

should be received over the course of the schedule execution there have to be two match list entries. The first entry is used only once and is deleted from the match list upon its first use. This entry matches an incoming message and places it in the appropriate shadow buffer. This match increments a counter to save the information that this message was already delivered. If the receive for this message is posted, the same counter is incremented again. So the second time this counter is increased we know that a) the message was received and b) that the corresponding receive has been posted. Therefore we can now copy that message from the shadow buffer into the user buffer. Only after the message has been copied we can resolve the outgoing dependencies of the receive, because the following operations might work on the data in the user buffer. Figure 7 shows how a portion of a cDAG schedule can be executed using Portals triggered operations, using an eager protocol. Figure 8 shows the automata which is applied to each vertex of the cDAG graph to translate its semantics into Portals API calls.

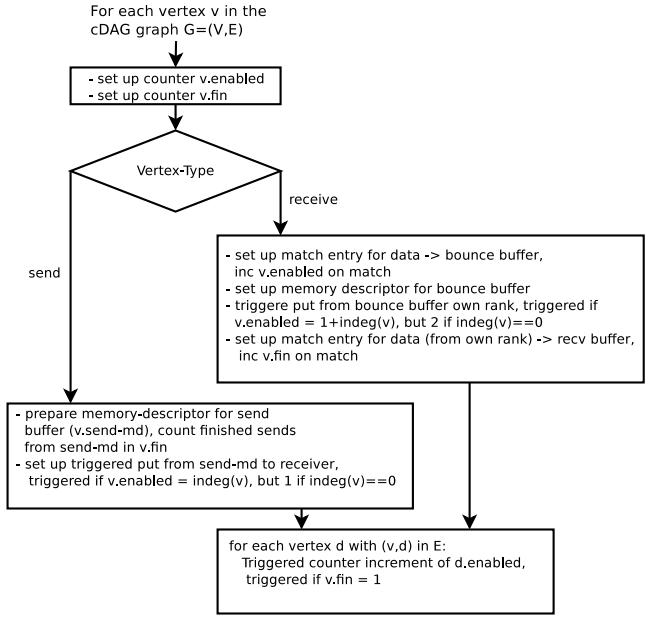

Fig. 8. Automata used to perform cDAG to Portals translation for the eager protocol

Unfortunately, Portals does not provide triggered operations to copy data from one memory descriptor to another. However, we can work around this problem by triggering a put to ourselves and having the second match list entry we mentioned before match only such self-sent puts.

V. Experimental Evaluation

The benchmarks in this section have been carried out on the Teller system at Sandia National Laboratories. Teller is a 104 node cluster of single socket AMD Llano Fusion processors. Each processor includes both a quad core 2.9 Ghz K10 (piledriver) processor and a 400 core 600 Mhz Radeon HD 6550D. For these experiments only the K10 processor was utilized. Each node has 16 GB of DDR3-1600 Mhz memory and nodes are connected via Qlogic QDR InfiniBand.

Currently no available network adapter supports Portals 4 triggered operations in hardware. However, the Portals 4.0 reference implementation [16] allows for the emulation of Portals specification compliant hardware. It supports the use of two different transports: OFED compatible network interfaces (i.e., InfiniBand, iWARP), and UDP. As Portals network interfaces provide for offloaded communication, the software reference implementation utilizes a progression thread spawned from each compute process. The progression thread polls the relevant event queue (an OFED EQ for IB and socket for UDP), and handles the reception of data and any subsequent triggered operations without the need for interaction with the compute thread. We are currently exploring alternative methods to provide progression without requiring a progress thread per every compute process.

We use the NBCBench [8] benchmark tool to compare the overlap of an implementation of a broadcast performed with cDAG-over-Portals with the one performed by cDAG-over-MPI, and we compare the non-overlappable part of the latency of the broadcast using our Portals implementation, as well as MPICH. Please note that the idea behind cDAG is not only to provide MPI collectives, but also to enable the formulation of any collective operation. We use MPI collectives for a comparison because they are some of the most well tuned complex communication operations available.

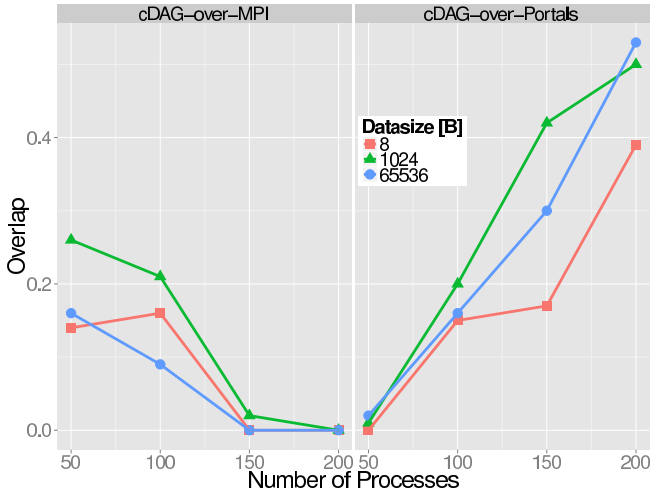

Fig. 9. Overlap for a Broadcast operation, performed with a cDAG interpreter which uses MPI as transport layer

NBCBench uses a work-loop, which is calibrated at the beginning of the benchmark, so that it takes the same time as the communication. To determine the overlap, the compute-loop is started after the (non-blocking) communication operation. By comparing the difference in execution time of the work-loop and the communication when run together compared to the not overlapped variant we can calculate which fraction of the communication is overlappable. A detailed description of this method can be found in [10].

In our first experiment, we compare the implementation of a broadcast in libNBC when using cDAG-over-MPI implementation, to a variant which uses cDAG-over-Portals, so only the low-level implementation of communication differs, both use the same communication topology for the broadcast. For this benchmark no manual progression, i.e., by repeatedly calling a test function, is performed. This is a realistic benchmark

for scientific codes, which often overlap communication with library calls, for example to the BLAS library. Inside of such a library progression of the communication is not performed, therefore our benchmark tests for true asynchronous progression. However, it is possible to achieve some overlap even with communication libraries that do not offer asynchronous progression, since they can overlap the first "round" of communication operations when the collective operation is started. This is what we observe in Figure 9. For small node-counts the MPI based variant of cDAG is able to get reasonable overlap, since the tested broadcast operation only needs a small number of rounds in this case. This decreases with larger node counts. In the case of Portals we observe the opposite. Due to the relatively high non-overlappable overhead of setting up all the counters, memory descriptors, match list entries, etc. for the collective operation the overlap is negligible for small node counts.

In Figure 10 we compare the non-overlapped part of the latency of broadcasts of different three different sizes implemented with cDAG-over-Portals to MPICH. While MPICH outperforms our Portals implementation for small data sizes (which is not surprising, since the software emulated Portals has a higher latency due to the emulation layer), Portals outperforms MPICH for large data sizes due to higher overlap.

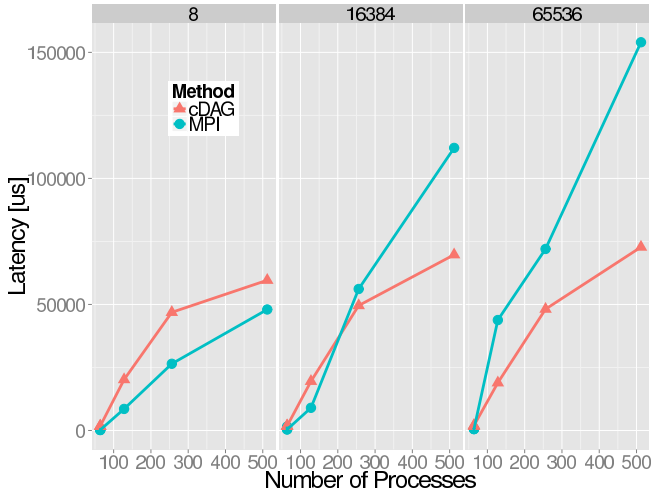

Fig. 10. Absolute time of the non-overlappable part of Broadcast operation performed using cDAG-over-Portals, compared to MPI

VI. Related Work

There has been much previous work on the topic of optimizing collective operations, for example [17]–[20]. When deciding which communication schemes to use for our cDAG collective implementation, we were guided by this body of work. Hardware acceleration of collective communication operation has been considered by many, the NEC Earth Simulator for example contained a hardware barrier implementation, however research has also been done to investigate if collective offload was possible in a more flexible manner, without a dedicated network. A lot of such work had been done by modifying the firmware of Myrinet NICs [21], [22]. Also the Quadrics Elan NIC was designed to run user code on the network processor and has been used to offload specific collective communication functions [23]. Some of the concepts used in Elan are very similar to the newest Portals API, for example Elan provided event functions which work similarly to Portals triggered operations.

The newest generation of InfiniBand [24] adapters from Mellanox provide the Collective Offload Resource Engine, CORE-Direct [25]. This adds a management queue to the standard InfiniBand queue pair. This management queue allows to delay certain operations until others are finished, and therefore to express dependencies between operations. Researchers reported positive results when implementing single collectives such as barrier and broadcast with this technology [26], [27]. To the best of our knowledge no one has attempted to show that the primitives offered by CORE-Direct are powerful enough to offload any communication schedule, or shown its limits. In [28] the authors define "building-blocks" for collectives, patterns such as 1-to-n send, or receive-and-replicate. With cDAG we provide a more fine-grained and complete abstraction than those patterns.

Offloading collectives is well suited to support non-blocking collectives, which are now a part of the MPI-3.0 standard [4], but have been in use before that, e.g., in the reference implementation libNBC [8]. The biggest problem when implementing non-blocking collectives without hardware acceleration is that one can not provide asynchronous progression easily [10] . This can be solved by spawning an extra thread which repeatedly progresses the communication by calling a test function (the approach used by the Portals 4 reference implementation) or forcing the user to split the computation used to overlap the collective in small chunks, so that the test function can be called between them (the approach taken by libNBC). Another option is to offload the collective into the OS, which can directly respond to network interrupts. This has been done in [29], [30], which ensures system wide asynchronous progress for non-blocking operations. However, in contrast to triggered operations it still leverages the host CPU to process incoming messages.

Collective offload with Portals 4 [3] has been shown for two versions of MPI_Allreduce in [2] and MPI_Barrier and MPI_Broadcast in [1]. In [14] the authors showed how to implement a rendezvous protocol with triggered operations. However, unlike the protocols described in this paper, this protocol can not be used to completely offload collective operations, as it requires the host CPU to make calls to Portals each time a receive is posted, where, for complete collective offload, the posting of a receive has to be done purely with triggered operations.

VII. Discussion

By attempting to design an abstract framework to offload arbitrary communication topologies we gained the following insights which should be taken into account by designers of network offload APIs, if their goal is to allow full offload of arbitrary communication topologies.

1.) The offload engine has to be able to dynamically (based on the completion status of previous send and receive operations) change the way incoming messages are handled (i.e., to which receive they are matched).

2.) Dynamic matching can be avoided, if messages are pre-matched, so that each message matches to exactly one

receive, however, such pre-matching requires synchronization, which make this approach impractical in certain scenarios. For example, implementing MPI-3 nonblocking collectives, as the MPI standard defines that neither calls to non-blocking collective functions, nor calls to the MPI_Test function are allowed to block.

3.) The offload engine has to provide a tag-space big enough for the tag-space which should be exported to the user and to discriminate between different collectives, as well as different instances of the same collective.

VIII. Conclusions

In this work we demonstrated how arbitrarily complex communication functions can be translated into a small set of offload-primitives by expressing them as a communication graph. We used the Portals 4.0 API as an example of such primitives. This approach proved to be valuable because it allows one to reason about the expressiveness of the offload primitives, and enables the building of complicated collective communication schedules based on a small set of protocols. By showing that certain protocols can not be implemented with the current Portals 4.0 API we were able to propose and implement a useful addition to this interface. We demonstrate that it is possible to achieve a high overlap using triggered operations, even with an emulated version of Portals where all operations are carried out on the host CPU.

References

[1] K. Hemmert, B. Barrett, and K. Underwood, "Using triggered operations to offload collective communication operations," *Recent Advances in the Message Passing Interface*, pp. 249–256, 2010.

[2] K. Underwood, J. Coffman, R. Larsen, K. Hemmert, B. Barrett, R. Brightwell, and M. Levenhagen, "Enabling flexible collective communication offload with triggered operations," in *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on*. IEEE, 2011, pp. 35–42.

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

[4] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 3.0," September 2012.

[5] W. Gropp, "Mpich2: A new start for mpi implementations," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 37–42, 2002.

[6] R. Graham, T. Woodall, and J. Squyres, "Open mpi: A flexible high performance mpi," *Parallel Processing and Applied Mathematics*, pp. 228–239, 2006.

[7] T. Hoefler, C. Siebert, and A. Lumsdaine, "Group operation assembly language - a flexible way to express collective communication," in *ICPP-2009 - The 38th International Conference on Parallel Processing*. IEEE, Sep. 2009.

[8] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for mpi," in *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[9] T. Hoefler and T. Schneider, "Optimization Principles for Collective Neighborhood Communications," Nov. 2012, accepted at SC12.

[10] 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.

[11] S. Pakin, "Receiver-initiated message passing over rdma networks," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[12] V. G. Cerf and R. E. Icahn, "A protocol for packet network intercommunication," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 2, pp. 71–82, 2005.

[13] J. Liu, J. Wu, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[14] B. Barrett, R. Brightwell, K. Hemmert, K. Wheeler, and K. Underwood, "Using triggered operations to offload rendezvous messages," *Recent Advances in the Message Passing Interface*, pp. 120–129, 2011.

[15] T. Hoefler and T. Schneider, "Runtime Detection and Optimization of Collective Communication Patterns," Sep. 2012, accepted at PACT 2012.

[16] Portals Development Team, "Portals 4.0 reference implemem," available at: http://code.google.com/p/portals4/ (Mar. 2013).

[17] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, p. 49, 2005.

[18] S. Sur, U. Bondhugula, A. Mamidala, H. Jin, and D. Panda, "High performance rdma based all-to-all broadcast for infiniband clusters," *High Performance Computing–HiPC 2005*, pp. 148–157, 2005.

[19] J. Träff and A. Ripke, "Optimal broadcast for fully connected networks," *High Performance Computing and Communications*, pp. 45–56, 2005.

[20] J. Bruck, C. T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 11, pp. 1143–1156, 2002.

[21] D. Buntinas, D. Panda, and P. Sadayappan, "Fast nic-based barrier over myrinet/gm," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*. IEEE, 2001, pp. 8–pp.

[22] D. Buntinas, D. Panda, J. Duato, and P. Sadayappan, "Broadcast/-multicast over myrinet using nic-assisted multidestination messages," *Network-Based Parallel Computing. Communication, Architecture, and Applications*, pp. 115–129, 2000.

[23] D. Roweth and A. Pittman, "Optimised global reduction on qsnet¡ sup¿ ii¡/sup¿," in *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*. IEEE, 2005, pp. 23–28.

[24] I. T. Association, *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[25] R. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, "Connectx-2 infiniband management queues: First investigation of the new support for network offloaded collective operations," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010, pp. 53–62.

[26] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, "Overlapping computation and communication: Barrier algorithms and connectx-2 core-direct capabilities," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010, pp. 1–8.

[27] M. G. Venkata, R. L. Graham, J. S. Ladd, P. Shamis, I. Rabinovitz, V. Filipov, and G. Shainer, "Connectx-2 core-direct enabled asynchronous broadcast collective communications," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011, pp. 781–787.

[28] H. Subramoni, K. Kandalla, S. Sur, and D. K. Panda, "Design and evaluation of generalized collective communication primitives with overlap using connectx-2 offload engine," in *2010 18th IEEE Symposium on High Performance Interconnects*, 2010, pp. 40–49.

[29] A. Nomura and Y. Ishikawa, "Design of kernel-level asynchronous collective communication," *Recent Advances in the Message Passing Interface*, pp. 92–101, 2010.

[30] T. Schneider, S. Eckelmann, T. Hoefler, and W. Rehm, "Kernel-based offload of collective operations - implementation, evaluation and lessons learned," in *Euro-Par (2)*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds., vol. 6853. Springer, 2011, pp. 264–275.