Exploiting Offload Enabled Network Interfaces

Salvatore Di Girolamo
ETH Zurich
digirols@inf.ethz.ch

Pierre Jolivet
ETH Zurich
pierre.jolivet@inf.ethz.ch

Keith D. Underwood
Intel Corporation
keith.d.underwood@intel.com

Torsten Hoefler
ETH Zurich
htor@inf.ethz.ch

***Abstract*—Network interface cards are one of the key components to achieve efficient parallel performance. In the past, they have gained new functionalities such as lossless transmission and remote direct memory access that are now ubiquitous in high-performance systems. Prototypes of next generation network cards now offer new features that facilitate device programming. In this work, various possible uses of network offload features are explored. We use the Portals 4 interface specification as an example to demonstrate various techniques such as fully asynchronous, multi-schedule asynchronous, and solo collective communications. MPI collectives are used as a proof of concept for how to leverage our proposed semantics. In a solo collective, one or more processes can participate in a collective communication without being aware of it. This semantic enables fully asynchronous algorithms. We discuss how the application of the solo collectives can improve the performance of iterative methods, such as multigrid solvers. The results obtained show how this work may be used to accelerate existing MPI applications, but they also display how these techniques could ease the programming of algorithms outside of the Bulk Synchronous Parallel (BSP) model.**

I. Introduction

The importance of interconnection networks is growing with the scale of supercomputers and datacenter systems. Machines with thousands to tens of thousands of endpoints are becoming common in large-scale computing. Communications become the major bottleneck in such machines be it to access shared storage, data redistributions (e.g., MapReduce), or communications in parallel computations. The most critical communication operations at scale are collective communications because they involve large numbers of processes, sometimes the whole system. Thus, network optimizations commonly focus on collective communications.

The steadily growing number of transistors per chip offers an opportunity to offload new capabilities to the network interfaces. For example, current high performance network interfaces support features such as lossless transport, remote direct memory access, and offloading for various network protocols such as TCP/IP. Programmable offload engines like we had in Quadrics Elan3/Elan4 are becoming progressively lower cost for network interfaces to include. First limited versions of such offload micro-architectures (MAs) are already available in Cray's Aries network [3] as well as Mellanox ConnectX2 [11].

Such offload features have been used to support the implementation of collective communications [23, 27, 28]. MPI-3 defines an extensive set of blocking and nonblocking as well as (user-defined) neighborhood collective communications. Previous works have either only supported partial offload requiring additional synchronization during setup or were limited to small message sizes. Thus, previous techniques cannot be used to implement *fully asynchronous offloaded* versions of all MPI-3 collective operations. Furthermore, existing protocols are specialized to particular NIC architectures.

In this work, we specify an abstract machine model for offload-enabled network interfaces. Our offload model captures common network operations such as send, receive, and atomics that can be executed by network cards. The execution of an offload program is advanced by events which can be either received messages or accesses from the host CPU. Using the offload model, we demonstrate how to design fully asynchronous offloaded collective operations for MPI-3. Furthermore, we demonstrate how Portals 4 can be used to implement this abstract model efficiently.

Our insights go far beyond the existing MPI-3 collective operations. Since each process in our offload model can start the execution of send, receive, and local operations at arbitrary processes, the model enables new semantics not offered by the current MPI specification. We show *solo collective communications*, an extended set of collective operations that can proceed and complete independently of other processes. Thus, solo collectives go one step further than MPI-3's nonblocking collectives which cannot complete (and often not proceed) until all processes have started the operation. Solo collectives, enabled by the offload model, will simply copy the buffer irrespective of the state of the owning process. This enables a powerful trade-off between data consistency (solo collectives may communicate outdated information) and process synchronization (solo collectives never wait). We show how this trade-off can be used to improve the performance of iterative algorithms, such as multigrid methods.

The specific contributions of our work are:

- The specification of an abstract machine interface for offload MAs with a proof of concept implemented on top of the Portals 4 reference library.
- Protocols for *fully asynchronous offloaded* MPI-3 collectives using arbitrary (optimized) communication schedules.
- A new class of *solo collectives* that allows trading off global synchronization cost for data consistency.
- A set of simulations showing the behavior of offloaded collectives at large scale.
- Microbenchmark results comparing offloaded and non-offloaded collectives.

II. Offload Enabled Architecture

Support for offloaded communications can vary dramatically from the dedicated hardware MAs infrastructure of the Cray Aries interface [10] to the programmable processors of the Quadrics network [20] to a dedicated core that can be associated with communications [1, 25]. The salient point is that all of these system architectures make communication operations

independent of the CPU performing application computation. We propose an abstract machine model and performance model in the context of such independence.

A. Abstract Machine Model for Offload MAs

In this section we introduce an abstract machine model describing the offload features offered by the next generation network cards. Our model considers two computational units: the CPU and the Offload Engine (OE). An offloaded operation is fully executed by the OE: CPU intervention is required only for its creation, offloading, and testing for completion.

In this model we define two main entities: communication and local computation. In both cases, they are defined as non-blocking *operations*. We adopt two-sided matching semantic in order to support complex communication schedules: processes are aware of the interactions among themselves. We use *send* and *receive* operations as data movement operations. An operation is created on the CPU and then offloaded to the OE.

A happens-before relation can be established between two operations a and b: we use the notation $a \rightarrow b$ to indicate that b can be executed only when a is completed. The definition of completion varies according to the type of operation. A *receive* is considered complete when a matching message is received. Differently, the completion of a *send* is a local event: it completes as soon as the data transmission is finished and the data buffer can be reused by the user. It is worth noting that, in our model, once the dependencies of b are satisfied, b can start without CPU intervention. Multiple dependencies can be handled with `AND` or `OR` policies. In the second case, the dependent operation can be executed when at least one of its dependencies is satisfied. We use the notation $(a_1 \wedge \ldots \wedge a_n) \rightarrow b$ or $(a_1 \vee \ldots \vee a_n) \rightarrow b$ for indicating an `AND` or an `OR` dependency between the operations $(a_1, \ldots, a_n)$ and b.

The life cycle of an operation is composed by the following states: *created*, if it has been created but not yet posted (i.e., offloaded); *posted*, the operation has been created and offloaded to the OE and *active*, if it is posted and it has no dependencies or all of them are satisfied. A *created* operation cannot be executed even if it has no dependencies or all of them are already satisfied; Moreover, an operation can be marked as:

- independent: if it can be activated as soon as it is posted;
- dependent: if it can be activated only when all its dependencies are satisfied;
- CPU-dependent: if it must be activated from the CPU.

A CPU-dependency can be installed even after the posting of an operation. It will have effect only if the operation is not yet executed. This allows to disable an operation, which can be re-enabled satisfying the installed dependency.

B. Performance Model for Offload MAs

Let x and y be two operations where $x \rightarrow y$, assume that x is a receive operation and it is the only dependency of y: once a message matching x is received, y must be executed. In order to make this step, the following sequence of events/actions must be handled: receive; matching; execution of y. In order to start the execution of y independently from the CPU, this entire sequence must be performed in an offloaded manner. This introduces the requirement that the message matching phase must be performed directly by the OE.

In order to catch this behavior, we introduce an additional parameter to the well-known LogGP [2] model. The standard parameters described by this model are: **L**: the latency parameter. It is defined as the maximum latency between two nodes in the network; **o**: the processor overhead. It is the time spent by a processor to send or receive a message; **g**: the gap between messages. It is defined as the minimum time interval between two consecutive message transmission or reception; **G**: the gap per byte. It models the time required by the NIC to send one byte; **P**: the number of processors.

The above parameters are not sufficient to model the matching phase that is now performed by the OE. A new corresponding parameter, called **m**, is introduced. It models the time needed to: 1) perform the matching phase; 2) satisfy the outgoing dependencies of the matched receive.

In our model the setup of an operation and its execution are decoupled: for example, a *send* can be installed at time t, paying the CPU overhead o at that time, but it could be effectively executed at a later time $\bar{t} \geqslant t$ when all its dependencies are satisfied. In general, the CPU overhead is accounted when the operation is installed by the host process.

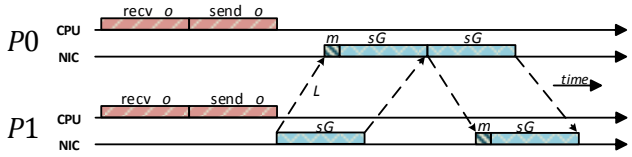


Fig. 1: Example showing a time-space diagram for the proposed performance model. $P1$ performs a send to $P0$. On $P0$, a send is scheduled to be executed as soon as the message from $P0$ is received.

Fig. 1 illustrates how the model can be applied to a ping-pong communication between the processes $P0$ and $P1$. As soon as $P0$ receives the message from $P1$ it responds with another message: this means that on $P0$ the sending of the "pong" message depends on the receive of the "ping". In our model, this dependency is handled and solved directly by the OE, without CPU intervention. The same behavior cannot be modeled in the LogGP model, since in that case we should count an additional o after receiving the "ping" message and before the sending the "pong" one. The overall cost T_{pp} of the ping-pong communication pattern in our model is:

$$T_{pp} = 2(o + L + sG + m)$$

The same pattern, in the LogGP model, has a cost of: $T'_{pp} = 2(2o + L + sG)$. The cost difference is explained by: a) differently from the LogGP model, the CPU overhead is decoupled from the actual execution of an operation, hence the overhead paid for the send at $P0$ can be overlapped with the one induced by the send at $P1$; b) in our model we have to take in account the matching phase cost, that is m.

C. A Case Study: Portals 4

Now we discuss how the proposed model can be applied on a concrete architecture, such as the one described by the Portals 4 specification [7]. This network programming interface is based on the one-sided communication model with the main difference that it does not use addresses to identify memory buffers on a remote node. A portal table is assigned to each network interface. Each entry of the portal table identifies three data structures: the priority list, the overflow list and the unexpected list. The first two lists provide entries describing

remotely accessible address regions, while the third is used for keeping track of unexpected messages.

Portals 4 supports two types of semantic: matching and non-matching. The first one has been introduced in order to better support tagged messaging interfaces, such as MPI. It allows the target node to add constraints to the list entries, that in this case are called match list entries (ME), such as the process ID that is allowed to access the described memory and a set of matching/ignore bits, acting like the MPI tag field. In order to map the computation model described in Sec. II-A, only the matching semantic is considered in this paper.

1) Communications: A target node exposes memory regions appending match list entries to the priority or overflow list. When a message arrives, the priority list is traversed searching for a matching list entry. In case no match is found, the overflow list is searched: if a matching ME is found there, the message header is inserted into the unexpected list. If no match is found neither in the overflow list then the message is dropped. The overflow and the unexpected list provide building blocks for handling unexpected messages: the user can provide "shadow" buffers appending list entries to the overflow list. When an ME is appended to the priority list, the unexpected list is searched for already delivered matching messages.

If a node (i.e., the *initiator*) wants to start an operation towards a *target* node, it has to specify a memory region using a memory descriptor (MD). If the operation is a *put* then the data will be copied from the buffer specified by the MD at the initiator to the one specified by the matching ME at the target. The *get* operation works in the opposite way: the data specified by the matching ME at the target will be copied into the buffer specified by the MD at the initiator.

2) Local computations: The Portals 4 specifications support atomic operations: one-way operations that take as operands the data specified by the MD at the initiator and the one described by the ME at the target. A local computation is a sequence of atomic operations with coinciding initiator and target node. This approach allows to offload simple local computations, enabling their asynchronous execution w.r.t. the CPU process.

3) Dependencies: Counters can be associated with memory descriptors and matching list entries. They are incremented each time a certain event is registered. Such events are related to operations performed on the associated data structures. We leverage this counting mechanism in order to detect the termination of outstanding operations. Portals 4 introduces the concept of triggered operations: we can associate an operation with a specific counter in a way such that it must be executed only when this counter reaches a certain threshold.

These two concepts (i.e., counters and triggered operations) can be used to map our dependency model. A counter is associated to each operation in order to detect its termination. If two operations x and y are defined in a way such that $x \rightarrow y$, then y is implemented as a triggered operation on the counter associated with x with a threshold equal to one: as soon as x will be completed and its counter will be incremented, then y will be triggered. Multiple dependencies can be implemented using an intermediate counter: if $x_i \rightarrow y$ with $i \in [1, \ldots, n]$, then a new counter ct_{xy} is created. When a x_i is completed, ct_{xy} is incremented by one. In this case y will become *active* only when ct_{xy} will reach a certain threshold, that will be n

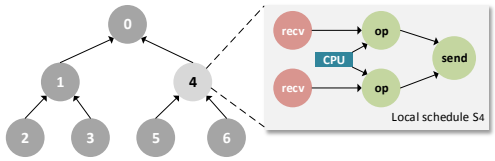


Fig. 2: Tree based reduce (left) and the local schedule executed by node 4 (right).

or 1 depending on whether the AND or the OR relation type is specified, respectively.

4) Operation Disabling: Portals does not allow to directly disable the execution of an operation. Suppose that a triggered operation b is targeting a buffer that a host process wants to modify. If the operation is already in execution, the buffer should not be modified by the host process. In the other case, we can disable the operation avoiding its triggering: if $a \rightarrow b$ and b is not yet executed, we can disable b decreasing the counter associated with a by one and setting b as to be triggered when such counter reaches at least the threshold of two. Even if a is executed, and hence its counter is incremented, b will not be triggered since its dependency counter (the one associated with a) has not reached the specified threshold. To re-enable b it is enough to increment its dependency counter: if a is already executed, then b will be immediately triggered; otherwise it will be executed as soon as a is completed.

5) Performance Model: The performance model discussed in Sec. II-B can be applied to a Portals 4 based architecture. In particular, we focus on the mapping of the parameter o and m, since the definition of L, g, and G is not altered. The o parameter accounts for the creation and the offloading of an operation: this corresponds to the creation of an ME or an MD and the interaction with the Portals 4 hardware, through which the operations can be offloaded to the OE. The time to perform the matching phase for an incoming message is captured by m. In Portals the matching phase consists in the searching of the priority list and, eventually, the overflow list.

III. Offloading Collectives

This section introduces offloaded collective communications such that the two following conditions are satisfied:

1) *No synchronization* is required in order to start the collective operation. Every process can start the operation without synchronizing or communicating with the others.
2) Once it has started, *no further CPU intervention* is required. The collective can complete without any CPU intervention.

A collective operation can be described as a directed graph, where a vertex is a participating node and an edge is a point-to-point communication. The ingoing edges represent receive operations, while sends are described by outgoing edges. The graph representing a binary-tree based reduce operation is shown on the left of Fig. 2. The set of operations executed by the internal node 4 and the dependencies among them (i.e., the schedule) are reported on the right.

Definition (Schedule)**.** *A schedule is a local dependency graph where a vertex is an operation, while an edge represents a dependency. It describes a partially ordered set of operations (i.e., point-to-point communications and local computations).*

Using the model proposed in Sec. II-A, an operation is defined *dependent* or *independent* according to its in-degree: zero ingoing edges means that the operation has no

dependencies and it can be immediately executed; an operation with an in-degree greater than zero can be executed as soon as all its incoming dependencies are satisfied. The completion of an operation leads to the satisfaction of all its outgoing dependencies. We define a schedule as complete when all the operations with out-degree equals to zero are completed.

Definition (Collective Communication)**.** *A collective communication involving n nodes is modeled as a set of schedules $S = S_1, \ldots, S_n$ where each node i participates in the collective executing its own schedule S_i.*

Offloading a collective operation means that every schedule S_i is fully executed by the OE of node i. This is possible only if the OE is able to handle all the components of a schedule (i.e., communications, local computations and dependency among them). The proposed abstract machine model catches them all, defining operations and dependencies as fully executed/handled by the abstract OE, hence allowing collective operation offloading.

A. Offloaded Point-To-Point Protocols

Collective operations are built on top of point-to-point communications, which we consider building blocks of our model. Two well-known protocols can be used, according to the message size, to address them in a correct and efficient way: the eager and the rendezvous protocol. The first one is used for small message sizes: it assumes that a receive buffer has already been posted at the destination node when the message from the sender arrives. When this assumption is not satisfied, the message is defined as unexpected. In order to handle such unexpected messages, shadow buffers can be provided requiring an additional copy at the time in which the receive buffer will be posted. However, since these buffers must have finite size, this protocol in not suitable for arbitrarily large message sizes. The rendezvous protocol is able to deal with arbitrary message sizes but it requires synchronization of the two involved nodes, introducing additional overheads.

1) Eager Protocol: We have to make a distinction between expected and unexpected messages. In the fist case, the message can be copied directly into the user-specified buffer. In the second, the message will be copied from the shadow to the user-specified buffer as soon as a matching receive will be posted. When the copy is completed, the shadow buffer can be re-used to catch other unexpected messages.

This protocol can be implemented with Portals leveraging the matching mechanisms provided by the priority and overflow lists (see Sec. II-C). The data copy from the shadow buffer to the user-specified one, not directly supported by Portals, can be implemented leveraging Portals *full events*. If an unexpected message is matched by an ME during the append phase, a proper full event will be raised allowing to handle the previously mentioned data copy. We assume that this process is race-free, meaning that the event will be generated at time $t \geq max(t_{OW}, t_{ME})$, where t_{OW} is the time at which the copy of the unexpected message on the shadow buffer is complete and t_{ME} is the time at which the matching ME is posted by the CPU. Please note that even if, in the unexpected message case, the data-copy must be performed by the CPU, the conditions (1) and (2) are still fulfilled: no synchronization is required and no CPU intervention is required after the creation of the operations. In fact, the potential overhead of the data copy due to unexpected messages is paid at the operation creation time.

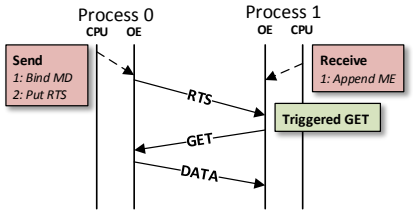


Fig. 3: Portals 4 implementation of the rendezvous protocol. After the RTS message is received by the target, a `GET` is triggered to perform the data movement.

2) Rendezvous Protocol: The rendezvous protocol is used for handling the transmission of arbitrarily large messages. It requires synchronization between the two communicating nodes. There are two variants of this protocol, differentiated by the node that initiate the protocol. In the sender-initiated version, a control message is sent to the receiver that will reply when the matching receive will be posted (and thus the receiver buffer will be ready). In the receiver-initiated version [22], the receiver has to signal to the sender when it is able to receive the message. Without loss of generality, in this work we consider only the sender-initiated variant of this protocol, since the receiver-initiated one can be implemented similarly.

In order to respect condition (1), we have to guarantee that no synchronization is required in order to start a collective operation. This implies that no processes synchronization can be required by the underlying point-to-point communications. The processes synchronization is considered as a side effect of the rendezvous protocol. However, this is no longer true if the entire protocol is fully offloaded since, in this case, its progression is totally independent from the host processes.

Fig. 3 sketches the Portals implementation of the rendezvous protocol. When a send operation is posted at the initiator node, a ready-to-send (RTS) control message is sent towards the target and an ME, let us call it ME_{data}, is appended to the priority list, in order to allow the target to get the data. The posting of a receive leads to the appending of an ME to the priority list (to catch the RTS) and the set up of a triggered `GET` in a way such that the data can be read from the send-buffer as soon as the RTS is received. The `GET` is a Portals operation, meaning that no addresses are required: the data will be read at the sender from the memory region specified by ME_{data}. If the RTS message is received as unexpected, the receiver-side protocol will start as soon as the receive will be posted.

IV. Solo Collectives

Traditionally, collective communications lead to the pseudo-synchronization of the participating nodes: at the end of the communication all the nodes have reached a point in which the collective call has been started. We refer this semantic as synchronized. In this section we propose a new non-synchronized semantic for collective operations, called *solo collectives*, in which the synchronization is completely avoided.

The idea of solo collectives is to globally start the operation as soon as one of the participating nodes (i.e., the initiator) joins the collective, independently from the state of the others. The main consequence of this approach is the relaxation of the usual synchronized semantic. A similar approach was proposed for UPC collectives by Ryne et al. [21]. However, they require that only one node is in charge to handle all the data movements,

leading always to a flat-tree virtual topology. In what follows, we how the previously discussed offloading principles can be used to enable solo collectives with arbitrary virtual topologies.

In order to execute a synchronized collective, each process i must create and execute, offloading the execution or not, its schedule S_i. Let us define $t = max_i(t_i)$ where t_i is the time at which process i starts the execution of S_i. A synchronized collective can be considered as concluded (i.e., all the S_i's have been executed) at a time $\bar{t}_s \geq t$. In a solo collective, each node creates and offloads its schedule S_i at time k. At that time the schedule is not executed, but only offloaded to the OE: a schedule in this state is defined as *inactive*. A node that wants to start the collective at time t_{init} activates its own schedule. The activation of a schedule leads to the broadcasting of a control message, necessary to activate the schedule of all the others nodes. A solo collective operation can be considered as concluded at time $\bar{t}_a \geq t_{init} + \epsilon$, where ϵ is the activation overhead that can be bounded with $\epsilon \leq max_i(\epsilon_i)$, with ϵ_i representing the time required to activate the schedule of node i. Please note that, differently from the synchronized case, $\bar{t}_a$ does not depend on any $t_i \neq t_{init}$.

Collective operations can be divided in two groups, according to the number of nodes contributing data to the computation of the final result. Let us define the nodes that participate to the collective providing data as active nodes, otherwise they are passive. The first group, defined as single-source collectives, contains the ones where only one node provides data (e.g., broadcast, scatter). Instead, in a multi-source collective all the nodes contribute data (e.g., reduce, gather, scan).

a) Single-Source Collectives: In this class of collectives there is only one active node. Other nodes have only to receive and eventually forward messages. There are two operations belonging to this class, that are broadcast and scatter. In both cases there is only one node, the root, that holds the data and wants to distribute it. In the broadcast case, the same message is sent to all the other nodes while, in the scatter, each node receives a personalized message. Let us consider a broadcast implemented with a binomial tree virtual topology. A similar discussion applies to the scatter. When an internal node receives a message from its parent, all the sends towards its children are activated, since their dependency is satisfied. This schedule can be expressed as:

$$S_k : recv \rightarrow send_i \quad \forall i \in C$$

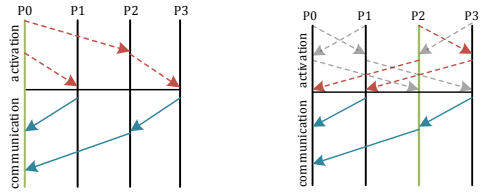
where C is the set of the children of node k. The activation of the non-root nodes coincides with the reception of the message from the parent, leading to an activation overhead ϵ of zero.

Using the performance model proposed in Sec. II-B and starting from the LogGP cost of this collective [14], we define the cost of a solo broadcast as:

$$T^{s-i}_{broadcast} = (L + m + \bar{s}G) \times \log_2(P)$$

where P is the number of nodes and $\bar{s}$ the message size. At each round we have one or more parallel communications with a cost of $L + m + \bar{s}G$ and there are $\log_2(P)$ rounds in total since the number of activated nodes doubles at each round.

b) Multi-Source Collectives: In multi-source collectives all the nodes are active: they receive, forward, and produce data necessary to the operation. In addition to rooted collectives like reduce and gather, all the unrooted collectives belong to this class. Let us consider the rooted operations first where we have two possible types of activation: root and nonroot. In the first case, the only node that can activate the collective is the root while, in the second, any node can be the initiator. In both cases the final target is to broadcast the activation message, what changes is the node that can start this broadcast.



(a) Root: P0; Initiator: P0 (b) Root: P0; Initiator: P2

Fig. 4: Root and non-root activation for the reduce operation. Activation messages are represented by dashed arrows. Gray dashed arrows are unused messages.

The root-activation can be implemented with any broadcast algorithm suitable for small message sizes. In Fig. 4 we propose to use a binomial tree virtual topology both for the activation and for the execution of the actual reduce operation. When a node is activated, it can immediately send its message to its parent or wait for children's data first, depending if it is a leaf or an internal node, respectively. In the examples reported in Fig. 4, $P1$ and $P3$ can send the message as soon as they are activated, since they are leaves. Node $P1$, instead, has to wait for $P2$ before sending to $P0$. The activation overhead ϵ is:

$$\epsilon = (L + m) \times \log_2(P)$$

if we assume a negligible activation message size (it does not carry any data in this case). The total cost of a solo reduce is:

$$T^{root}_{reduce} = (2L + 2m + \bar{s}G + \omega) \times \log_2(P)$$

where $\bar{s}$ is the message size, P is the number of involved nodes and ω is the cost of applying the reduce operator. Please note that even if this cost is higher than the respective synchronized reduce cost, here we are relaxing the assumption that all the involved nodes join the collective at the same time.

The nonroot activation allows any node to start the collective, meaning that we have P possible broadcast activation trees. A naive approach to implement such type of activation could consist in the set up of P different schedules for each node: one for each possible activation tree in which it could be involved. However, this approach is not scalable, since it would require $\mathcal{O}(P)$ space for each node. The solution is to use a slightly modified version of the recursive doubling algorithm, exploiting the fact that the virtual topology described by this algorithm can be reduced to a binomial tree if we consider only the communications generated by one single node. In the original algorithm each node x sends to and receives from a node y, where y is a node with a distance that doubles at each step: the next step (i.e., the next send and receive pair) can be executed when the receive of the previous one is concluded:

$$recv_l \rightarrow send_{l+1}$$

Since we want to activate only the communications belonging to the binomial tree rooted in the initiator, we require that:

$$(recv_0 \vee \ldots \vee recv_l) \rightarrow send_{l+1}$$

meaning that as soon as one receive is concluded, all the subsequent sends must be executed. An example of non-root

activation is reported in Fig. 4b, where a reduce operation towards the node $P0$ can be started by any node. Since the activation can be done in $\log_2(P)$ steps, the activation overhead ϵ is the same of the root-activation case, leading to the same collective communication cost of the root activation case.

In the non-root activation there is the possibility that more than one node try to activate the same collective. However, multiple activations of the same collective must not lead to multiple executions of the schedule or part of it by some nodes.

Claim. *In the nonroot activation, if $k > 1$ nodes try to activate the same collective, then it will be executed exactly once.*

The proof of the previous claim is trivial. Consider the worst-case in which all the nodes activate the same collective at the same time: all the communications described by the recursive-doubling algorithm will take place, leading to the $\log_2(P)$ receptions of the activation message by each node. However, the schedule of the collective will start as soon as the first activation message is received: subsequent activation requests will not affect the execution of already activated operations.

A. Multi-Version Scheduling

In the previous section we introduced solo collectives. They allow a node to start and complete a collective operation without requiring the explicit join of all the other involved nodes. The only requirement is that each node has its own schedule posted at a time before its actual activation. The pre-posting is not required for correctness, which would violate condition (1) of Sec. III. It is a requirement to guarantee the asynchronous activation of the schedules: if a node x starts an operation and another node y has not posted yet its own schedule, then the collective will progress at y when that schedule will be posted.

Let us consider the case in which the same collective can be executed multiple times and, each time, it can be initiated by a different node. We need a mechanism to support the execution of multiple solo collectives: the multi-version schedule. It allows the pre-posting of k versions of a collective, where a version is a schedule of the same collective, potentially targeting different data buffers. We define three possible states of a posted schedule: *disabled*, meaning that all the contained operations cannot be executed; *enabled*, if it is ready to be executed; *in use* if at least one operation has been completed (e.g., a message is received or sent). A multi-version schedule can be described as a FIFO queue of schedules: at each time there is only one schedule that is enabled. When the enabled schedule becomes *in use*, then the next in the queue is enabled. The enabling of the next schedule in the queue can be offloaded to the OE. If the schedule S_{i-1} precedes S_i in the multi-version queue, we can describe their relation as:

$$\bigvee_{op_k \in I_{i-1}} op_k \rightarrow op_j \quad \forall op_j \in I_i$$

where I_{i-1} and I_i are the sets of the locally independent operations of the schedules S_{i-1} and S_i, respectively. An operation is locally independent, w.r.t. its own schedule, if it does not depend on any operation belonging to the same schedule. Initially S_i is disabled, since all its locally independent operations cannot be executed: they depend on the ones contained in I_{i-1}. They will be enabled as soon as one of the independent operations of S_{i-1} will have been completed (it is an `OR` dependency), leading to the activation of S_i.

B. Data Consistency

The introduction of solo collectives leads to two questions: 1) How to prevent race conditions between the CPU and the OE? 2) How to manage data buffers in case of multiple outstanding versions of the same solo collective? The MPI specification establishes that the ownership of a buffer is passed to the library until the operation targeting it is complete. This approach, that solves the problem raised in question one, cannot be adopted in our case. A solo collective is posted at process P_i at a time that precedes the one at which the operation will be effectively activated by P_i itself. Moreover, it could be executed before the reaching of that point. In order to let the CPU and the OE to synchronize, we adopt an on-demand ownership scheme: it is passed to the OE at the time in which the solo collective is instantiated and the host process can ask for temporary ownership. If the currently enabled schedule is not in use, then it will be disabled and the ownership will be granted. Otherwise the process will have to wait until its completion.

Now let us discuss the problem raised by question two. Having a multi-version schedule of the same single-initiator collective means that a process is able to support up to k asynchronous executions of the same collective. There are two different policies that can be applied for handling data-buffers.

- *Single buffer*. The same buffer is associated with all the versions of the schedule: it will be always updated with the last known value.
- *Pipelined buffers*. A different buffer is associated with each version of the schedule.

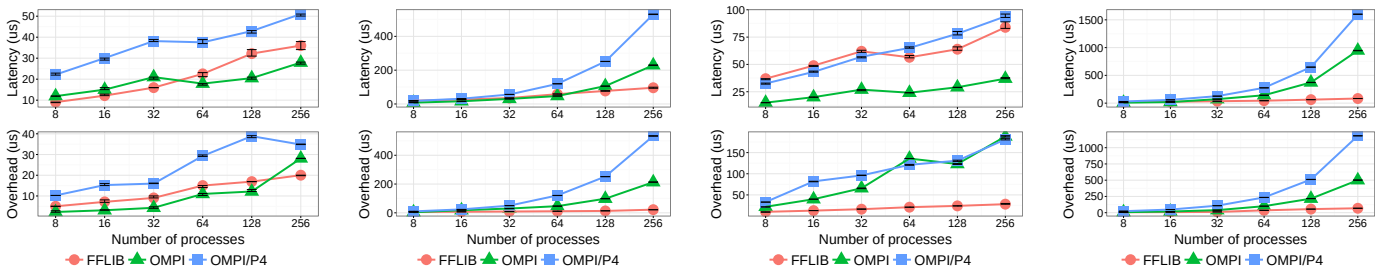
These two policies can be applied either to the send or the receiver buffer, and they lead to different semantics of the same collective. Considering the receive buffer, the only difference is if to overwrite its content or not with new data. In the first case, the data that the process was unable to receive will be discarded. Applying the *pipelined buffers* policy to the send buffer, instead, limits the asynchronicity of the solo collectives, since it makes CPU-dependent all the operations targeting it.

C. Use Case: Multigrid

Multilevel preconditioners are one of the dominant paradigms in contemporary large-scale partial differential equations simulations [6, 24]. While they are in theory optimal methods requiring $\mathcal{O}(N)$ work if N is the size of the discrete system to solve, in practice, they can incur significant communication and synchronization overheads.

We consider a two-grid hierarchy solver where the coarsening is carried out by only one process. Hence, we have a communication scheme involving a gather and a scatter: for each iteration, the root has to gather the data, perform the coarsening, and scatter the results. The workers have to perform their computation and participate in collective operations. Similar coarsening schemes are typical in the domain decomposition community, cf. [17]. To fully exploit the potential of communication offloading, an asynchronous iteration scheme may be used [4, 9]. In a typical multigrid method, using asynchronous iterations inside the smoother is likely to worsen the convergence rate, but because of higher concurrency, it can still be beneficial in terms of time to solution, for example when using accelerators [5].

We now demonstrate a simple benchmark where the above described computation/communication pattern is carried



(a) Broadcast (b) Scatter (c) Allreduce (d) Allgather

Fig. 5: Non-blocking collectives latency/overhead. We report median over 100 iterations. The 95% CI is within the 5% of reported medians. The message size is 50 B.

out for 100 iterations. We simulate solo and synchronized collectives in a highly imbalanced scenario: Each process injects random noise with a duration in the interval $[0, I]$ in each iteration; $I \approx 100ms$ is the iteration time. Solo collectives improve the completion time by a factor of ≈ 1.5 on our test system (described in Sec. V). We attribute this to the missing synchronization overheads and the fact that the data is gathered and scattered from the root in a "one-sided" fashion, enabling to fully overlap computation and communication by the workers and the coarsening process. We remark that only the described simple communication pattern is taken into account and not how the convergence rate is affected from the adoption of a fully asynchronized approach, which deserve further investigation outside the scope of this work. We expect that significant improvements are possible, especially at extreme scales.

V. Experimental Result

Results were obtained on Curie, a Tier-0 system for the Partnership for Advanced Computing in Europe composed of 5,040 nodes made of 2 eight-core Intel Sandy Bridge processors. The interconnect is an InfiniBand QDR full fat-tree [19]. In our experiments, we used OpenMPI version 1.8.4 as MPI implementation, compiled with two different backends: InfiniBand (OMPI) and Portals 4 (OMPI/P4). The obtained results are compared against FFLIB[1], a proof of concept library that we built on top of the Portals 4 reference library (P4RL) [7], implementing the concepts described by the proposed abstract machine model. The P4RL was compiled with InfiniBand support. In all experiments, two threads are assigned to each MPI process in order to minimize the overhead induced by the auxiliary thread used by Portals for the NIC emulation.

A. Offloaded Collectives

In this experiment we compare offloaded and non-offloaded collectives showing two measurements: latency and overhead. The latency is defined as the maximum finishing time of a collective among all the nodes. We report this value due to its impact on the parallel running time of load-balanced applications [12]. The overhead, instead, is the fraction of communication time that cannot be overlapped with computation. For each communicator size, we report the median among 100 samples.

In Fig. 5c the latency/overhead comparison for the allreduce collective operation is showed. The adopted algorithm is the binomial-tree based one, consisting of two phases: 1) reduce towards a designated root; 2) the root broadcast the computed result. The OMPI latency is roughly a factor of two lower respect to FFLIB. This is due to the presence of an additional software layer introduced by the P4RL. This is confirmed by the results of OMPI/P4: the same algorithm/implementation is a factor of 2 slower when the P4RL is used as backend. The overhead introduced by FFLIB is the time necessary to create and offload the schedule to the OE, which grows with the number of scheduled operations (that is logarithmic in the number of processes for the allreduce algorithm). As expected, the overhead introduced in the non-offloaded case has a larger multiplicative constant. This is explained by the fact that parts of the schedule (i.e., communication rounds) require CPU intervention in order to be executed. Fig. 5a shows the results for the broadcast operation. Since this is a one-round collective, the broadcast is a low-overhead operation. In this case, FFLIB gets an higher overhead w.r.t. OMPI, due to the relatively high cost of interfacing the P4RL. On the other side, the offloaded execution allows to reach latencies comparable to the ones reported for OMPI. The algorithms employed by OMPI for non-blocking scatter and allgather are linear in the number of processes, while FFLIB implements these two collectives with binomial and recursive doubling algorithm, respectively. In both cases they have a logarithmic cost in the number of processes. This explain the results of Fig. 5b and Fig. 5d.

B. Simulations

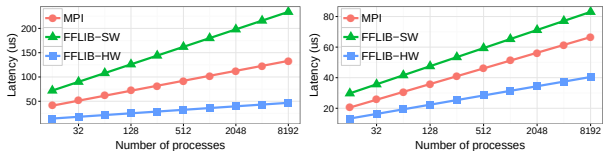
In this experiment we study and compare offloaded and non-offloaded collectives at large scale. We extended the LogGOPSim [15] simulator with the Portals 4 semantic and the proposed performance model (see Sec. II-B) in order to simulate offloaded operations. Table I reports the LogGP parameters measured on the target machine with the Netgauge [13] tool. The m parameter measures the average time taken by the P4RL to complete the matching phase for each received message.

	L	o	g	G	m
OMPI	$2.7\mu s$	$1.2\mu s$	$0.5\mu s$	$0.4ns$	-
OMPI/P4	$5\mu s$	$6\mu s$	$6\mu s$	$0.4ns$	$0.9\mu s$

TABLE I: LogGP parameters for OMPI and OMPI/P4

As expected, the values of OMPI/P4 are larger than OMPI. This is due to the additional software layer needed to emulate NIC offload functionalities. The idea is to simulate a non-software emulated NIC using the OMPI LogGP parameters (which has only one software layer directly interfaced to the NIC), and setting $m = 0.3\mu s$, which is the simulation parameter used by Underwood et al. [26] to model the incoming message processing time of a Portals 4 based NIC. The simulations for allreduce and broadcast are reported in Fig. 6. In both cases we report the non-offloaded version (MPI) using the OMPI LogGP parameters and the offloaded version using the OMPI/P4 LogGP parameters (FFLIB-SW). In addition, we show offloaded collectives simulation with the above discussed LogGP parameters combination (FFLIB-HW). The simulation show that the current software-based offloaded collectives

[1] http://spcl.inf.ethz.ch/~digirols/fflib.tar



(a) Allreduce (b) Broadcast

Fig. 6: Collective operations simulation. The message size is 50 B.

have the worst latency. On the other side, fixing the LogGP parameters, offloaded collectives get an improvement of ≈ 3 and ≈ 1.4 for the allreduce and the broadcast, respectively.

VI. Related Work

There exist a number of past works concerning the delegation of communication, either point-to-point or collective, to an external processing unit. Graham et al. [11] investigate the offloading of collective operation exploiting InfiniBand management queues provided by ConnectX-2 HCA (Host Channel Adapter). Different versions of HCA-based broadcast are proposed in [27, 18], as well for the allgather [16, 28]. However, this solution allows only round-based design of collective communication algorithms. Subramoni et al. [23] propose a set of offloaded collective communication primitives, showing how different algorithms for collective operations can be derived from a different composition of such building blocks. In this case, the algorithmic design is limited to the prosed building blocks, while our approach has a finer grain since it allows to model single basic operations. There is also a number of works related to the exploiting of the concepts introduced by the Portals 4 network interface, however they do not fully satisfy the conditions discussed in Sec. III. Schneider et al. [22] discuss about protocols for fully offloaded collectives, however, their protocol requires synchronization among the involved nodes. Barrett et al. [8] propose an offloaded version of the rendezvous protocol based on Portals 4 triggered operations, requiring CPU intervention in the unexpected message case. Even in that case synchronization is required in order to set up the data structures that are needed to the correct progression of the communication. Underwood et al. [26] propose an approach for the collective communication offloading. However, it does not fully implement commonly used protocols able to guarantee correctness and efficiency of point-to-point communications, on top of which collective operations are built. As a consequence, none of these works allow to implement offloaded collective operations respecting, at the same time, the MPI specifications.

VII. Conclusion

The aim of this work is to propose an abstract machine model for offload MAs, and a relative performance model, that is able to catch the offload functionalities of next generation network cards. We exploited the proposed MA proposing solo collective operations that allow to avoid implicit collectives synchronization. We implemented a library, FFLIB, that is proof-of-concept of the proposed MA, comparing offloaded collectives performance to the ones provided by the best-performing MPI implementation w.r.t. the target machine. In the end, we showed large-scale simulations, discussing the potential implication of a hardware implementation of the Portals 4 specifications.

Acknowledgments

This work was granted access to the HPC resources of TGCC@CEA made available within the Distributed European Computing Initiative by the PRACE-2IP. The authors want to thank Ryan Grant, Roberto Belli and the SPCL group for the useful discussions. P. Jolivet has been supported by an ETH Zürich Postdoctoral Fellowship.

References

[1] N. Adiga and et. al. An Overview of the BlueGene/L Supercomputer. In *Proc. of the 2002 ACM/IEEE Conf. on Supercomputing*, 2002.
[2] A. Alexandrov, M. F. Ionescu, K. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proc. of the 7th annual ACM Symp. on Parallel Algorithms and Architectures*, 1995.
[3] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. *Cray XC series network*, 2012.
[4] H. Anzt, E. Chow, and J. Dongarra. Iterative sparse triangular solves for preconditioning. *21st International European Conference on Parallel and Distributed Computing*, 2015.
[5] H. Anzt, S. Tomov, M. Gates, J. Dongarra, and V. Heuveline. Block-asynchronous multigrid smoothers for GPU-accelerated systems. *Procedia Computer Science*, 9, 2012.
[6] A. Baker, R. Falgout, T. Kolev, and U. M. Yang. Scaling *hypre*'s multigrid solvers to 100,000 cores. In *High-Performance Scientific Computing*. 2012.
[7] B. Barrett, R. Brightwell, R. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. Maccabe, and T. Hudson. *The Portals 4.0.2 Network Programming Interface*, 2014.
[8] B. Barrett, R. Brightwell, S. Hemmert, K. Wheeler, and K. Underwood. Using triggered operations to offload rendezvous messages. In *Recent Advances in the Message Passing Interface*. 2011.
[9] D. Chazan and W. Miranker. Chaotic relaxation. *Linear algebra and its applications*, 2(2), 1969.
[10] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray Cascade: A Scalable HPC System Based on a Dragonfly Network. In *Proc. of the 2012 ACM/IEEE Conf. on Supercomputing*, 2012.
[11] 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 *Proc. of the 10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, 2010.
[12] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proc. of the 2007 ACM/IEEE Conf. on Supercomputing*, 2007.
[13] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. Netgauge: A network performance measurement framework. In *High-Performance Computing and Communications*, volume 4782 of *Lecture Notes in Computer Science*. Springer, 2007.
[14] T. Hoefler and D. Moor. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Journal of Supercomputing Frontiers and Innovations*, 1(2), 2014.
[15] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim: Simulating Large-scale Applications in the LogGOPS Model. In *Proc. of the 19th ACM Int. Symp. on High Performance Distributed Computing*, 2010.
[16] G. Inozemtsev and A. Afsahi. Designing an Offloaded Nonblocking MPI_Allgather Collective Using CORE-Direct. In *2012 IEEE International Conference on Cluster Computing*, 2012.
[17] P. Jolivet, F. Hecht, F. Nataf, and C. Prud'homme. Scalable domain decomposition preconditioners for heterogeneous elliptic problems. In *Proc. of the 2013 ACM/IEEE Conf. on Supercomputing*, 2013.
[18] K. Kandalla, H. Subramoni, J. Vienne, S. Raikar, K. Tomko, S. Sur, and D. Panda. Designing Non-blocking Broadcast with Collective Offload on InfiniBand Clusters: A Case Study with HPL. In *19th IEEE Symp. on High-Performance Interconnects*, 2011.
[19] C. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 100(10), 1985.
[20] D. Roweth and A. Pittman. Optimised Global Reduction on QsNet-II. In *13th IEEE Symp. on High-Performance Interconnects*, 2005.
[21] Z. Ryne and S. Seidel. Ideas and specifications for the new one-sided collective operations in UPC, 2005.
[22] T. Schneider, T. Hoefler, R. Grant, B. Barrett, and R. Brightwell. Protocols for fully offloaded collective operations on accelerated network adapters. In *42nd Int. Conf. on Parallel Processing*, 2013.
[23] H. Subramoni, K. Kandalla, S. Sur, and D. Panda. Design and evaluation of generalized collective communication primitives with overlap using connectx-2 offload engine. In *18th IEEE Symp. on High-Performance Interconnects*, 2010.
[24] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler. Parallel geometric-algebraic multigrid on unstructured forests of octrees. In *Proc. of the 2012 ACM/IEEE Conf. on Supercomputing*, 2012.
[25] S. R. W. Timothy G. Mattson, David Scott. A TeraFLOPS Supercomputer in 1996: The ASCI TFLOP System. In *Proc. of the 1996 Int. Parallel Processing Symp.*, 1996.
[26] K. Underwood, J. Coffman, R. Larsen, S. Hemmert, B. Barrett, R. Brightwell, and M. Levenhagen. Enabling flexible collective communication offload with triggered operations. In *19th IEEE Symp. on High-Performance Interconnects*, 2011.
[27] W. Yu, D. Buntinas, and D. K. Panda. High-Performance and Reliable NIC-Based Multicast over Myrinet/GM-2. In *32nd Int. Conf. on Parallel Proc.*, 2003.
[28] W. Yu, D. Buntinas, and D. K. Panda. Scalable and High-Performance NIC-Based Allgather over Myrinet/GM. *IEEE Cluster Computing*, 2004.