Exploiting Offload Enabled Network Interfaces

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

Pierre Jolivet
CNRS
pierre.jolivet@enseeiht.fr

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

Torsten Hoefler
ETH Zurich
htor@inf.ethz.ch

***Abstract*—The authors propose an abstract machine model for offload-enabled architectures. The Portals 4 network interface is used to implement the proposed abstract model. They propose the concept of persistent offloaded operations that can reduce creation and offloading overheads. The presented results show how this work can be used to accelerate existing MPI applications.**

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 those in Quadrics Elan3/Elan4 are becoming progressively less expensive for network interfaces to include. First limited versions of such offload microarchitectures are already available in Cray's Aries network [2] as well as Mellanox ConnectX2 [8].

Such offload features have been used to support the implementation of collective communications [15, 16]. For example, 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 network-in-chip (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. Events such as received messages or accesses from the host CPU can advance the execution of an offload program. Using the offload model, we demonstrate how to design fully asynchronous offloaded collective operations for MPI-3. Furthermore, we demonstrate in a case study how Portals 4 can be used to implement this abstract model efficiently, discussing how this interface can be extended in order to optimize cases presenting a high reuse of network operations (that is, persistent operations).

The proposed abstract machine model is implemented in *FFLIB*, a programming abstraction library built on top of the Portals 4 reference library. Using this implementation we benchmark different applications in order to show the potential effects of hardware offloading and, in particular, of the fully asynchronous progression of collective operations.

II. Offload-Enabled Architecture

Support for offloaded communications can vary dramatically, from the dedicated hardware microarchitecture's infrastructure of the Cray Aries interface [6] to the programmable processors of the Quadrics network [13], to a dedicated core that can be associated with communications [12]. The salient point is that all of these system architectures make communication operations independent of the CPU performing the computation. We propose an abstract machine model and performance model in the context of such independence.

A. Abstract Machine Model for Offload Microarchitectures

In this section, we introduce an abstract machine model describing the offload features offered by the next-generation network cards. Our model comprises 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 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 has been copied into the receive buffer. Instead, 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. Arbitrary Boolean expressions can be defined to express multiple dependencies. We use the notation $(a_1 \wedge \ldots \wedge a_n) \rightarrow b$ or $(a_1 \vee \ldots \vee a_n) \rightarrow b$ to express an `AND`

or `OR` dependency between the operations $(a_1, \ldots, a_n)$ and b, respectively.

An operation's life cycle comprises the following states: *created*, if it has been created but not yet posted (that is, offloaded); *posted*, if 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, or *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 an effect only if the operation is not yet executed. This lets the host process disable an operation, which can be re-enabled later by satisfying the installed dependency.

On some offload hardware, on which the OE/NIC is distant from the CPU, downloading operations can have a significant overhead. Usually, operations are consumed during their execution. To mitigate the downloading overhead, we propose a mechanism to cache operations at the OE. These operations will automatically be de-activated once they execute and can be re-activated by the CPU later. Because persistent operations are not automatically removed after they execute, they occupy OE resources until the host process removes them. In order to tackle this aspect, a software-controlled cache can be implemented at the OE and managed by the CPU. When the downloading of a new operation fails due to missing resources in the OE, a cache replacement mechanism can be executed by the CPU in order to free OE resources.

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. To make this step, the following sequence of events and actions must be handled: receive, matching, and execution of y. Because the execution of y must start independently from the CPU, this entire sequence has to be carried out by the OE. This introduces the requirement that the message matching phase must be performed directly by the OE.

To catch this behavior, we introduce an additional parameter to the well-known LogGP model [1]. The standard LogGP parameters are: latency **L**, defined as the maximum latency between any two nodes in the network; processor overhead **o**, that is the time spent by a processor to send or receive a message; gap between messages **g**, which models the minimum time interval between two consecutive messages; gap per byte **G**, which is the time required by the NIC to send one byte; and the number of processors **P**.

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

In our model, an operation's setup and 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 for when the operation is installed by the host process.

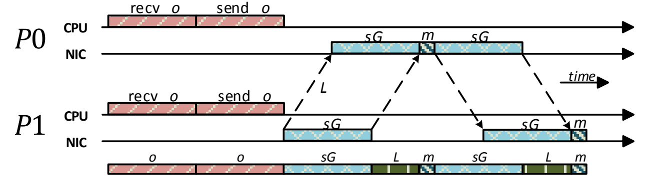


Fig. 1: Time-space diagram of the proposed performance model. $P1$ sends to $P0$ a message of size s. On $P0$, a send of a same-size message is scheduled to be executed as soon as the message from $P1$ is received.

Figure 1 illustrates how the model can be applied to a ping-pong communication of a message with size s between $P0$ 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, because in that case we should count an additional o after receiving the "ping" message and before sending the "pong" one. The overall cost T_{pp} of the ping-pong communication pattern in our model is:

$$T_{pp} = 2(o + L + s \cdot G + m)$$

The same pattern, in the LogGP model, has a cost of: $T'_{pp} = 2(2o + L + s \cdot G)$. The cost difference is explained by the fact that, unlike in the LogGP model, in our model the CPU overhead is decoupled from the actual execution of an operation, so the overhead paid for the send at $P0$ can be overlapped with the one induced by the send at $P1$. In addition, in our model we must account for the matching phase cost, which is m.

C. Portals 4 Case Study

We can apply our proposed model on a concrete architecture, such as the one described by the Portals 4 specification. 9 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 portal table entry 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, whereas the third tracks unexpected messages.

Portals 4 supports two types of semantic: matching and nonmatching. The first one has been introduced to better support tagged messaging interfaces, such as MPI. It lets the target node add constraints to the list entries, which in this case are called match list entries (MEs), such as the process ID that is allowed to access the described memory and a set of matching/ignore bits, which act like the MPI tag field. In order to map the computation model, we consider only the matching semantic in this article.

1) Communications: A target node exposes memory regions by appending match list entries to the priority or overflow list. When a message arrives, the priority list is traversed and searched by the OE for a matching list entry. If no match is found in the priority list, 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, the message is dropped. The overflow and the unexpected list provide building blocks to handle unexpected messages: the user can provide "shadow" buffers by 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 (that is, the *initiator*) wants to start an operation toward a *target* node, it must specify a memory region using a memory descriptor (MD). If the operation is a *put*, 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 specification supports atomic operations: they 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 can be expressed as a sequence of atomic operations with a coinciding initiator and target node. This approach lets us offload simple local computations, which enables their asynchronous execution with regard to the CPU process.

3) Dependencies: Counters can be associated with memory descriptors and matching list entries, counting occurrences of specific events related to such data structures (that is, operation completion). We leverage this counting mechanism to detect the termination of outstanding operations. Portals 4 introduces the concept of triggered operations, which are associated with a specific counter that must be executed when such a counter reaches a certain threshold.

These two concepts (that is, counters and triggered operations) are used to map our dependency model. A counter is associated with 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 is completed and its counter is incremented, y will be triggered. Multiple dependencies can be implemented using an intermediate counter: if $x_i \rightarrow y$ with $i \in [1, \ldots, n]$ and $n > 1$, then a new counter ct_y is created. When an x_i is completed, ct_y is incremented by one. In this case, y will become *active* only when ct_y reaches a certain threshold, which can be n or 1 depending on the specified relation type: `AND` or `OR`, respectively.

4) Operation Disabling: Portals 4 does not allow to directly disable the execution of an operation. Suppose that a triggered operation b is targeting a buffer that the initiator 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 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 reenable 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) Persistent Operations: The current Portals 4 interface specification does not support persistent operations. To enable their support, the interface should be extended to offer two new features: auto-resetting counters and permanent triggered operations. An auto- resetting counter is a normal counter with an associated reset threshold: the counter is set to a default value (such as zero) when the reset threshold is reached. A persistent triggered operation is a triggered operation that is not consumed once it is executed. It will be triggered every time the associated counter reaches the specified threshold.

Consider two operations, x and y, such that $x \rightarrow y$ and no other dependencies are set on y. Moreover, y is defined as a persistent operation. The dependent operation (y) is implemented as a persistent triggered operation to be executed when the counter ct associated with x reaches the threshold of 1. The counter ct has a reset threshold of 1: as soon as x is completed, y is activated and ct is reset. When ct will be incremented again, due to a new completion of x, the operation y will be reexecuted. It is worth noting that this mechanism does not depend on the nature of x but on the counter associated with the MD or ME that x is targeting. If ct is associated with an MD, then x can be a Portals put, get, or atomic. It will be incremented as soon as the operation is completed. Otherwise, if ct is associated to an ME, it will be incremented as soon as an operation targeting that specific ME will be completed.

A minimal extension of the Portals 4 interface to support persistent operations would consist of introducing a persistency option for triggered operations and extending the counter interface to introduce the reset threshold. Portals 4's resource management allocates a fixed amount of resources to each process. This mechanism can be easily extended to support the management of permanent triggered operations.

6) Performance Model: The performance model discussed in Sec. II-B can be applied to an architecture based on Portals 4. In particular, we focus on the mapping of the parameters o and m, because the definitions of L, g, and G are not altered. The o parameter accounts for an operation's creation and offloading: 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 4, the matching phase consists of searching the priority list and, eventually, the overflow list.

III. Offloading Collectives

Here, we introduce fully offloaded collective communications such that the following two 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 to complete the collective.

A collective operation can be described as a directed graph, in which vertices are operations (either communications or computations) and edges are dependencies among them. There are two types of dependencies: intra- and inter-node. An inter-node dependency can be established between a send and a receive operation: the matching of the receive leads to the satisfaction of the dependency. Intra-node dependencies are

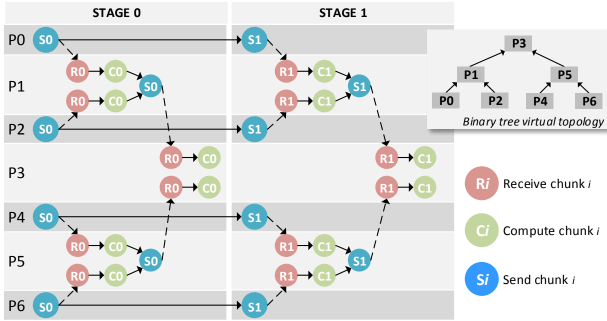


Fig. 2: Dependency graph of a two-stages pipelined, binary-tree based reduce. Inter- and intra-node dependencies are represented by dashed and continuous arrows, respectively. Operations are represented by labelled circles indicating the operation type (R: Receive, S: Send; C: Computation) and the pipeline stage.

between operations on the same node, as described by the proposed abstract machine model. Figure 2 shows a graph representing a pipelined binary-tree-based reduce operation. The example represents a two-stage pipeline. The leaves just have to send a chunk of data at each pipeline stage to their parent. Internal nodes ($P1$, $P5$) have to apply the reduce operator to both the messages received by their own children, sending the computed intermediate result to the parent node $P1$. The order between the stages of the pipeline is enforced by intra-node dependencies between the operations at the leaf nodes and message tags. A *schedule* is the set of operations and intra-node dependencies at each node.

Definition (Schedule)**.** *A schedule is a local dependency graph in which a vertex is an operation, whereas an edge represents an intra-node dependency. It describes a partially ordered set of operations (that is, point-to-point communications and local computations).*

Using the model proposed earlier, an operation is defined as *dependent* or *independent* according to its in-degree: zero ingoing edges means that the operation has no dependencies and can be executed immediately; 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 consider a schedule as complete when all the operations with out-degrees equal to zero are completed.

Definition (Collective Communication)**.** *A collective communication involving n nodes is modeled as a set of schedules $S = S_1, \ldots, S_n$ in which 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 can handle all of a schedule's components (that is, communications, local computations, and dependencies among them). The proposed abstract machine model catches them all, defining operations and dependencies as fully executed and 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 are the building blocks of our model. We can use two well-known protocols, according to the message size, to address them correctly and efficiently: eager and rendezvous. The eager protocol 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 and is copied into a shadow buffer, requiring an additional copy at the time in which the receive buffer will be posted. However, because these buffers must have a finite size, this protocol is not suitable for arbitrarily large message sizes. The rendezvous protocol can deal with arbitrary message sizes at the cost of introducing additional synchronization overhead. Although the eager protocol leads the receiver to synchronize on the sender, the rendezvous protocol implies the full synchronization of the involved nodes. Apparently, both protocols violate condition (1). However, if the full protocol is offloaded to the OE, the synchronization is totally decoupled from the host process, thus preserving the mentioned condition.

1) Eager Protocol: We have to distinguish between expected and unexpected messages. In the first case, the microarchitecture copies the message directly into the user-specified buffer. In the second case, the message will be copied from the shadow to the user-specified buffer as soon as a matching receive is posted. Once this copy is completed, the shadow buffer can be reused to catch other unexpected messages.

This protocol can be implemented with Portals leveraging the matching mechanisms provided by the priority and overflow lists. The data copy from the shadow buffer to the user-specified one, not directly supported by Portals, can be implemented by leveraging Portals' *full events*. If an unexpected message is matched by an ME during the append phase, a proper full event will be raised. Such an event can be used by the host process to trigger the data copy from the shadow to the user-provided buffer. 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.

2) Rendezvous Protocol: This 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 initiates the protocol. In the sender-initiated version, a control message is sent to the receiver, who will reply when the matching receive will be posted (and thus the receiver buffer will be ready). In the receiver-initiated version [14], the receiver must signal the sender when it is able to receive the message. Without loss of generality, in this article we consider only the sender-initiated variant of this protocol; the receiver-initiated one can be implemented similarly.

Figure 3 sketches the Portals implementation of the rendezvous protocol between two processes $P0$ and $P1$. The send operation at $P0$ appends an ME (say, ME_{data}) to the priority list and sends a ready-to-send (RTS) control message toward

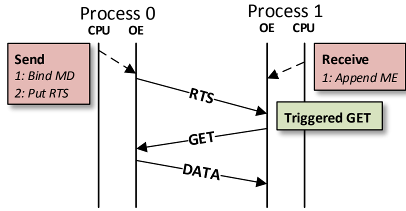


Fig. 3: Portals 4 implementation of the rendezvous protocol. After the target receives the RTS message, a GET is triggered to perform the data movement.

$P1$. When the receive is activated at $P1$, an ME is appended to the priority list to catch the RTS. Moreover, a get operation is installed in a way such that the data can be read from $P0$ as soon as the RTS is receivedhence, the get is dependent on the RTS reception. If the GET is a Portals operation, the initiator $P1$ does not have to specify remote memory addresses: the data will be read from the memory region specified by ME_{data} at the target $P0$. If the RTS message is received as unexpected, the receiver-side protocol will start as soon as the receive is posted.

B. Schedule Caching

Applications that repeatedly issue a fixed set of collective operations (that is, the same parameters) can benefit from schedule caching at the OE. Normally, the cost of starting an offloaded collective operation comprises both the cost of the algorithm used to produce the schedule and the cost of offloading the scheduled operations.

Schedule caching can be implemented by exploiting persistent operations. We define a schedule's "frontier" as the set of independent operations contained in it. This set's cardinality is indicated with n_I. Because a cached schedule must be explicitly activated by the host process, all the independent operations are converted to dependent ones installing CPU dependencies (independent operations are executed as soon as they are posted). All of a cached schedule's operations are defined as persistent. The cost of starting a collective operation with a cached schedule corresponds to the cost of satisfying n_I CPU dependencies, which is the cost necessary to activate the schedule's frontier.

It is worth noting that the impact of schedule caching on the overall application performance depends on the frequency with which cached collective schedules are executed and on the single operation creation and offloading cost.

C. Hardware Provisioning

We defined the OE as a generic entity to which the CPU can offload the execution and the progression of communications and computations. The main advantages of a hardware-implemented OE are low latency, immunity to OS noise, and zero host CPU usage. One limitation of this approach is the fixed amount of resources on the OE. Considering a Portals 4-compliant OE, in order to address this problem, we must answer the following questions: how many triggered operations do we need, and how many counters?

As we discussed earlier, triggered operations and counters are used to implement dependencies among operations. Considering a parallel application and limiting our analysis to collective operations, we define the amount of resources

Collective	Algorithm	Time	MD/ME	Triggered Ops.	Counters	Memory
Broadcast	Linear	$O(P)$	1	0	0	S
	Binary Tree	$O(log(P))$	2	2	1	S
	Binomial Tree	$O(log(P))$	2	$log(P)-1$	1	S
All-to-All	Linear	$O(P)$	2	0	0	S
	Recursive Doubling	$O(log(P))$	2	$log(P)-1$	1	$P \cdot S$
All-Reduce	Binomial tree	$O(log(P))$	$log(P)$	$2 \cdot log(P)$	$log(P)$	S
	Recursive Doubling	$O(log(P))$	$log(P)$	$2 \cdot log(P)$	$log(P)$	S
Scatter	Linear	$O(P)$	1	0	0	$P \cdot S$
	Binomial Tree	$O(log(P))$	2	$log(P)-1$	1	$P \cdot S$
Gather	Linear	$O(P)$	1	0	0	$P \cdot S$
	Binomial Tree	$O(log(P))$	2	1	1	$P \cdot S$
Reduce	Linear	$O(P)$	P	P	P	S
	Binomial Tree	$O(log(P))$	$log(P)$	$log(P)$	$log(P)-1$	S

TABLE I: Maximum Portals 4 resources occupation per endpoint of various collective operation algorithms. (P: communicator size; S: message size.)

needed at the OE as a function of the number of dependent operations in a schedule, the number of outstanding collective operations, and the number of processes sharing the same OE. In addition, offloaded data-movement operations are considered for the operations count. This suggests that the algorithms used to implement collective operations play an important role in the resource usage at the OE. In particular, we notice a tradeoff between OE resource utilization and the performance of such algorithms (that is, latency). For example, reducing the fan-out of a distribution tree will lead to a longer tree and then likely to a higher latency, but it will also reduce the number of dependent operations (that is, the number of children) - hence, the resources usage at the OE. Table I reports the time complexity [11] and the maximum resource occupation among all the involved nodes of different possible implementations of collective operations. The completion time of these operations is expressed according with the LogGP model.

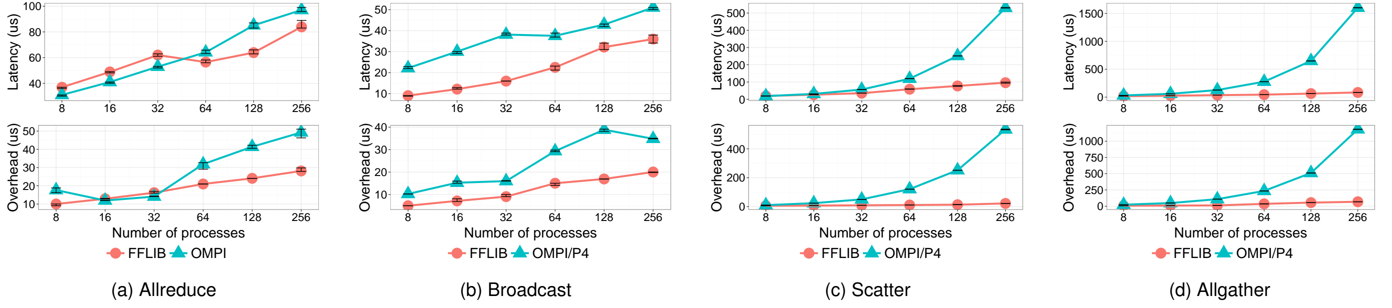
IV. Experimental Result

Results were obtained on Curie, a Tier-0 system for the Partnership for Advanced Computing in Europe composed of 5,040 nodes made of two eight-core Intel Sandy Bridge processors. The interconnect is an InfiniBand QDR full fat-tree. In our experiments, we use OpenMPI version 1.8.4 compiled with two back ends: InfiniBand (OMPI) and Portals 4 (OMPI/P4). We compared our results against FFlib, a proof-of-concept library that we built on top of the Portals 4 reference library (P4RL) [4]. Such a library implements the concepts described by the proposed abstract machine model. The P4RL was compiled with InfiniBand support. In all experiments, we scheduled one MPI process per node and two hardware cores per MPI process to minimize the overhead induced by the auxiliary thread used by Portals for the NIC emulation.

In order to report a fair comparison, we compare only OMPI/P4 results. If OMPI is directly interfaced to the Infiniband network cards through the Verbs interface, collectives can be executed faster (up to 2.5 times faster). Comparing OMPI/P4 with FFlib lets us assess the benefits of OE architectures directly.

A. Collective Operations Latency/Overhead

In this experiment, we compare offloaded and nonoffloaded 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 because



(a) Allreduce

(b) Broadcast

(c) Scatter

(d) Allgather

Fig. 4: Non-blocking collectives latency and overhead with $50\,\mathrm{B}$ message size.

of its impact on the parallel running time of load-balanced applications [10]. 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. The 95 percent confidence interval (CI) is always within the 5 percent of the reported medians

Figure 4a shows the latency and overhead comparison for the AllReduce collective operation. The adopted algorithm is the binomial-tree-based one, consisting of two phases: a reduce toward a designated root followed by the broadcast of the computed result. Although the FFlib and OMPI/P4 latencies of this collective operation are comparable, OMPI/P4 presents an overhead up to 1.8 times higher with regard to FFlib when using 256 processes. This is explained by the fact that the nonoffloaded approach of OMPI/P4 requires CPU intervention to execute parts of the schedule (that is, communication rounds). The overhead introduced by FFlib, instead, is just the time necessary to create and offload the schedule to the OE, which grows with the number of scheduled operations (which is logarithmic in the number of processes for the AllReduce algorithm). Figure 4b shows the results for the broadcast operation: FFlib shows an improvement up to a factor of two for both latency and overhead. The algorithms employed by OMPI/P4 for non-blocking scatter and AllGather are linear in the number of processes, whereas FFlib implements these two collectives with the binomial and recursive doubling algorithms, respectively. In both cases, they have a logarithmic cost in the number of processes. This explains the results of Figures 4c and 4d.

B. *Micro-Benchmarks*

We evaluated the effects of introducing fully offloaded collectives with two microbenchmarks: parallel 3D Fast Fourier Transformation (3DFFT) and the Pipelined Generalized Minimal Residual Method (PGMRES). The reported results are the medians as a function of the communicator size and configuration (that is, FFlib and OMPI/P4). The 95 percent CI is always within 5 percent of the reported values.

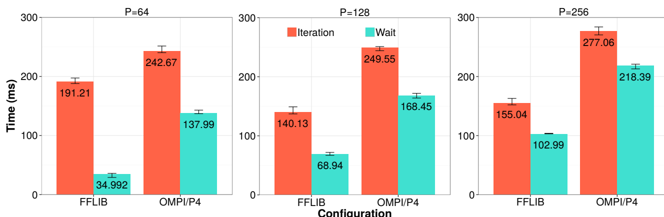


Fig. 5: The Pipelined Generalized Minimal Residual Method (PGMRES) benchmark. The iteration time and wait time are given for both the examined configurations and for different communicator sizes (P).

In the PGMRES experiment, we solve the 2D Laplace equation on a Cartesian grid using the example number 46 from the Portable, Extensible Toolkit for Scientific Computation (PETSc) distribution [3]. We perform executions with 64, 128, and 256 MPI processes. The global preconditioner is an additive Schwarz method [5], which involves sparse collectives operations at each iteration (halo exchanges). The PGMRES method is the chosen Krylov method [7]. Unlike in the standard GMRES, there are no blocking global collectives at each iteration for computing dot products. Instead, a nonblocking MPI_Iallreduce is initiated at the end of each iteration, and its completion is checked after the computation of the next preconditioner-matrix-vector product. We modified the PETSc distribution to allow switching between MPI and FFlib collectives. Figure 5 reports the results of the PGMRES experiment. The test executes a maximum of $1,000$ iterations, and the problem size (3200×3200 grid size) is chosen in a way such that this limit is reached for all the tested communicator sizes. The high wait time can be explained by communications among subgroups of processes taking place at each iteration: these can introduce load imbalance among the processes, leading to an increase of the synchronization time implied by the global reduce operation. We can observe how the fully offloaded configuration presents a speedup of waiting time up to 2.4 times in the $P = 128$ case, which suggests that such a configuration is less sensible to the discussed load imbalance.

In the parallel 3D Fourier transform, the data is transformed in the x and y direction first and, after applying a parallel transposition (that is, all-to-all communication), the transformation is applied in the z direction. Following the approach proposed by Torsten Hoefler and colleagues [9], we implement the z-plane communication in a nonblocking, pipelined manner to overlap the communication of the i-th block with the computation of the $(i+1)$-th one. Figure 6 shows the pipeline stage completion time and the wait time for communicator sizes of 64, 128, and 256 MPI processes. For each communicator size, the results of FFlib and OMPI/P4 are reported. A single pipeline stage comprises the following steps: compute the i-th block, wait for the completion of the $(i-1)$-th block communication, and set up the i-th block communication.

The problem size is selected such that the total number of iterations is approximately $1,000$. The results show how the introduction of fully offloaded collectives leads to a speedup in the waiting time, up to 3.11 times in the $P = 256$ case. This improvement lets FFlib keep scaling until $P = 256$, whereas OMPI/P4 stops at $P = 128$.

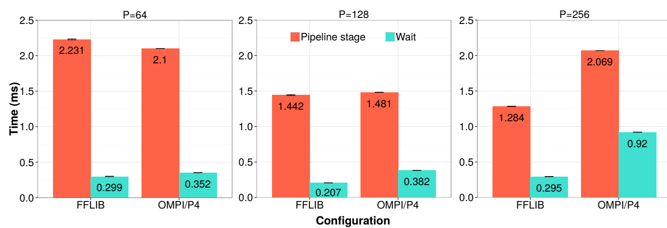


Fig. 6: 3DFFT benchmark. The pipeline stage completion time and wait time are given for both the examined configurations and for different communicator sizes.

V. Future Work

Future work can enhance the discussed model to introduce Turing completeness. In this way, complex schedules can be offloaded to the OE, enabling their asynchronous execution with respect to the host. Persistent operations are the first step toward this extension: they can be installed on the OE and can be executed multiple times without requiring any further host intervention. A next step could enrich the model and introduce conditional branching, discussing how existing architectures, like Portals 4-compliant ones, can be extended to support such an extended model.

Acknowledgments

This work was supported by Intel under the OCoOl project. It was granted access to the HPC resources of TGCC@CEA made available within the Distributed European Com- puting Initiative by the PRACE-2IP. Pierre Jolivet has been supported by an ETH Zürich Postdoctoral Fellowship.

Authors' Biographies

Salvatore Di Girolamo is a PhD student at ETH Zürich. His research interests include high-performance networking and communication offload for next-generation networks and RMA programming. Di Girolamo received an MSc in computer science and networking from the University of Pisa and Scuola Superiore Sant'Anna, Italy. Contact him at digirols@inf.ethz.ch.

Pierre Jolivet is a scientist at CNRS in the Toulouse Institute of Computer Science Research. His research interests include numerical linear algebra, computational physics, and high-performance scientific computing. Jolivet received a PhD in applied mathematics from the Université de Grenoble. Contact him at pierre.jolivet@enseeiht.fr.

Keith D. Underwood leads a product architecture team for next-generation host fabric interfaces in the Omni-Path Architecture group at Intel. His research interests include hardware/software co-design to support HPC networking, enhanced collective operation architectures, and alternative computing architectures. Underwood received a PhD in computer engineering from Clemson University. Contact him at keith.d.underwood@intel.com.

Torsten Hoefler is an assistant professor in the Department of Computer Science at ETH Zürich. His research interests include perform- ance-centric software development, scalable networks, parallel programming techniques, and performance modeling. Hoefler received a PhD in computer science from Indiana University. Contact him at htor@inf.ethz.ch.

References

[1] 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.
[2] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. Cray XC Series Network. *Cray Inc., White Paper WP-Aries01-1112*, 2012.
[3] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. PETSc web page. http://www.mcs.anl.gov/petsc, 2015.
[4] 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.
[5] V. Dolean, P. Jolivet, and F. Nataf. *An Introduction to Domain Decomposition Methods.* Society for Industrial and Applied Mathematics, 2015.
[6] 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.
[7] P. Ghysels, T. Ashby, K. Meerbergen, and W. Vanroose. Hiding Global Communication Latency in the GMRES Algorithm on Massively Parallel Machines. *SIAM Journal on Scientific Computing*, (35), 2013.
[8] 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.
[9] T. Hoefler, P. Gottschling, and A. Lumsdaine. Brief Announcement: Leveraging Non-blocking Collective Communication in High-performance Applications. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'08*, pages 113–115. Association for Computing Machinery (ACM), 06 2008. (short paper).
[10] 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.
[11] T. Hoefler and D. Moor. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Journal of Supercomputing Frontiers and Innovations*, 1(2), 2014.
[12] N.R. Adiga et. al. An Overview of the BlueGene/L Supercomputer. In *Proc. of the 2002 ACM/IEEE Conf. on Supercomputing*, 2002.
[13] D. Roweth and A. Pittman. Optimised Global Reduction on QsNet-II. In *13th IEEE Symp. on High-Performance Interconnects*, 2005.
[14] 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.
[15] 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.
[16] W. Yu, D. Buntinas, and D. K. Panda. Scalable and High-Performance NIC-Based Allgather over Myrinet/GM. *IEEE Cluster Computing*, 2004.