Simulation of the Portals 4 protocol, and case study on the BXI interconnect

Julien Emmanuel, Matthieu Moy, Ludovic Henrio, Grégoire Pichon

▶ **To cite this version:**

Julien Emmanuel, Matthieu Moy, Ludovic Henrio, Grégoire Pichon. Simulation of the Portals 4 protocol, and case study on the BXI interconnect. HPCS 2020 - International Conference on High Performance Computing & Simulation, Dec 2020, Barcelona, Spain. pp.1-8. hal-02972297

HAL Id: hal-02972297
https://hal.science/hal-02972297v1

Submitted on 20 Oct 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Simulation of the Portals 4 protocol, and case study on the BXI interconnect

Julien Emmanuel*†, Matthieu Moy*, Ludovic Henrio*, Grégoire Pichon†

* Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
first.last@ens-lyon.fr

† Atos, Échirolles, France
first.last@atos.net

***Abstract*—We present a new network simulator, which models the Portals 4 communication protocol used in High Performance Computing (HPC). It is built on top of SimGrid and uses cooperative actors to model the interactions between compute nodes in a supercomputer. Unlike most simulators in HPC, it models both communications on the interconnect and on the PCIe network inside each compute node, whithout going for a full emulation of the hardware. The simulator can be used to optimize or debug an application without having to use an actual supercomputer. This is made possible by leveraging SimGrid's flow model and it enables accurate simulation with good performances, even when running the model on a laptop. We test this simulator with custom experiments as well as existing Portals code, and compare the results with Portals executions on an actual cluster using Atos' BXI interconnect.**

I. Introduction

Because of physical limitations, it becomes increasingly difficult to design better CPUs, which is why today's supercomputers improve their performance by augmenting the number of compute nodes. These nodes need to be able to communicate as fast as possible in order to use the full potential of CPUs and accelerators, using interconnection networks as performant as possible. Even though many efforts are made to design hardware that allows faster communications on the high-speed network (Section II gives an overview of mechanisms commonly used for this purpose), the interconnect is still a bottleneck for the performance of most HPC applications. Application developers must allow parallelism between computation and communication to fully exploit the capacities of the hardware. The workloads generated by these applications can be very diverse, and modern interconnects are generally made of complex hardware, which makes the performance of an application on a specific cluster hard to predict, and the optimal configuration difficult to determine. In particular, the interconnect needs to provide mechanisms that guarantee the reliability of communications, but an inadequate configuration of such mechanisms can cause a loss in performance. Influencing the behaviour of the interconnect to improve performance can also be done at the "fabric" level (i.e. the configuration of the whole cluster on management machines), using mechanisms such as adaptive routing for example, which makes studying the performance of an interconnection network even harder. This is why designing performant yet accurate simulators to model Network Interface Controllers (NIC) and switches is still an active research topic. Because of the complexity of the hardware, and the growing number of nodes in a typical cluster, such a model has to make a compromise between speed and accuracy. This choice depends mainly on the problem that the simulator needs to solve: for example, if the model will be used to make decisions when designing new hardware, it must be as accurate as possible otherwise some effects will not be detected, but if the goal is to determine the performance of an application across a whole cluster, the model needs to trade some accuracy for speed (otherwise an even bigger cluster will be needed to run the simulation). Our study's final aim is to be able to model a realistic application running on a whole cluster (potentially thousands of nodes), because the diversity and complexity of communication patterns in existing scientific applications don't allow us to obtain accurate results by studying the behaviour of only a few machines. Our simulator should therefore be able to help developpers of scientific applications in the optimization and debugging of their code, by allowing them to understand what limits the performance of their application, and how the interconnect might be configured to make the execution faster.

In this paper we present a cooperative actor-based simulator which models the Portals 4 protocol [1], specified by Sandia National Laboratories and commonly used in HPC. This simulator is especially suited to model the BXI interconnect [2], which provides a hardware implementation of Portals 4 and is used in some of the best european supercomputers. Our simulator has the following features:

- it enables the online simulation of Portals code, without any need for pre-existing traces,
- the simulation of several nodes can be achieved on a regular laptop, thanks to SimGrid's [3] flow model,
- the model accounts for PCIe transfers between the NIC and the RAM of each machine, which have durations of the same orders of magnitude as interconnect transfers,
- the platform's configuration can be tuned to predict the

performance of NICs with different characteristics.

The paper is organized as follows: Section II presents the context regarding supercomputers' interconnect and existing simulators, Section III explains the design of our simulator, Section IV shows a validation of our model by comparing its predictions with executions on a real cluster using the BXI interconnect, and Section V concludes the paper.

II. Related Work

In the HPC field, manufacturers of interconnection networks use different protocols to ensure an optimal performance and quality of service. The most famous are InfiniBand [4][5], uGNI, DMAPP and Portals [1]. InfiniBand is mainly implemented by Mellanox' hardware, uGNI and DMAPP are APIs exposed by Cray's interconnects [6] (for generic communications and distributed memory applications respectively), and Portals is currently implemented by Bull's interconnect (BXI) [2]. Older versions of Portals were implemented in the XT series of Cray's supercomputers (XT3 to XT5) through their Seastar interconnect [7] as well as in the ASCI Red supercomputer.

Our work focuses on Portals 4: while there are already several studies and simulators for InfiniBand [8] and Cray's interconnect [9], there is (to our knowledge) no available model of the Portals 4 API, although it is a standard protocol represented by the BXI interconnect in the Top500 ranking [10].

Portals, as well as most other HPC protocols, allows remote direct memory accesses (RDMA) between compute nodes. It is reliable, connectionless (unlike TCP for example), and it enables a good overlap between computation on the CPU and communication on the NIC. This last feature comes from the offloading of the network processing on the NIC, which allows the CPU to be involved as little as possible in these data transfers. Portals semantics allow for zero-copy communications: data can be streamed to (or from) user's memory without being stored in intermediate buffers, along with OS-bypass: no system call is needed to communicate on the network, to save CPU time.

Along with performance, the interconnect must guarantee a good quality of service through end-to-end reliability (E2E), fabric management, and allow efficient implementations of common HPC APIs, such as the message passing interface (MPI), partitioned global adress space (PGAS) models, etc.

Existing simulators of supercomputer's interconnect use various methods and models, which allow for different tradeoffs between precision and execution speed.

We can differentiate two main types of simulators: online and offline. An online simulator runs an application step by step on a virtual platform (which is usually described in a configuration file given as an input to the model) to analyze it, whereas an offline simulator replays a trace obtained in an execution of the application on an actual platform. Even though running an offline simulation is usually faster, it has a few disavantages: a supercomputer must be available to obtain the trace that the simulation requires, which limits the ability to model future platforms, that are not yet usable. Moreover, trace files can be extremely large due to the number of nodes in a supercomputer, which makes them difficult to store and analyze. On the other hand, online simulators allow more realistic predictions for clusters that are not available, and are generally easier to run since they don't require any prior execution of the application.

While some simulators, like the SMPI interface [11] in Simgrid, are able to run both online and offline simulations, the majority of them are designed to support only one of those methods. Because we want to be able to explore a range of configurations of the interconnect as wide as possible, and to make prediction for any cluster easily, we focus on online simulation in order not to have to deal with traces.

Existing network models have different levels of precision: some of them, such as ns-3 [12], OMNeT++ [13] or SST/macro's packet models [14], model each network packet individually. This makes them very precise, but also orders of magnitude slower than real-life executions of the considered application. These simulators are useful to emulate the behaviour of a few nodes, but they cannot be used to model efficiently a whole cluster running a complex application, because of the amount of time that it would require.

On the other hand, some simulators choose a simpler model, which allows better scaling, and enables the simulation of thousands of nodes. For example, the LogP is a family of analytical models commonly used to make scalable simulators such as LogGOPSim [15] or analytical models in SST/macro [14]. The downside of this family of models is that to be as fast as possible they use simplistic approximations to estimate the duration of network transfers. In particular, these heuristics don't take some effects into account, such as congestion, although it can have a great impact on the performance of a cluster.

Finally, some models try to be more complete while staying faster than packet-level simulators. For example, in this category, SimGrid [3] uses a flow-based model at message-level to account for congestion in the network. This is achieved by using an analytical model (similar to the LogP family), which handles congestion by solving a *Max-Min* optimization problem: the sum of all activities on a specific resource (such as a network link or CPU) cannot exceed the maximum capacity for this resource, while each activity must get a resource share as big as possible. So if no priority is set for any activity, each one sharing the same resource will get an equal share (of bandwidth for links, or CPU for compute activities).

Models for existing HPC protocols (such as Portals) are often very precise, to capture every detail of the protocol and of the processing in the hardware, at the cost of slow executions. In this category, Mellanox provides an OMNeT++ model of their InfiniBand interconnect [16] and studies have

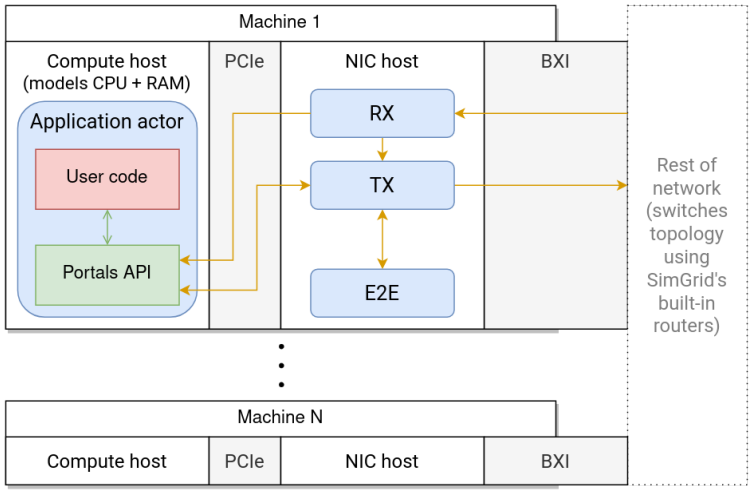

Fig. 1. Actors placement on the simulated hosts

been conducted to model accurately the OmniPath interconnect [17]. Unfortunately, running a realistic application on such a simulator is orders of magnitude slower than an execution on a real cluster, which makes optimizing the application through simulation very hard (if not impossible).

On the other hand, some simulators (such as SMPI [11]) model communications at MPI-level, and apply fixed factors to links' bandwidth and latency depending on message size to account for the specificities of protocols at the lower level and of the underlying hardware. The problem with this technique is that tuning the simulator to find optimal parameters is challenging, and it isn't as precise as modeling the real behaviour at a lower level.

Since the goals of our study is to tune low-level configuration parameters depending on high-level communication patterns, we chose to model communications at message-level (which is orders of magnitudes faster than packet or flit level), but to account for every message issued by the NIC, including Portals' and E2E's acknowledgements (ACK ; more details on this mechanism are given in Section III-D), and for PCIe transfers between the NIC and the CPU (or RAM). This makes the simulation precise enough for our needs while being able to model a whole cluster, and it allows users to tune Portals' behaviour through simulation.

III. Using SimGrid for Portals simulation

The simulator that we present is built on top of SimGrid, and aims at modeling Portals behaviour. We use SimGrid because it allows us to build scalable simulations while modeling congestion on the network. For our needs the simulator has been tuned to model the BXI interconnect, which offers a hardware implementation of Portals, but our work can be extended to other Portals implementations. In this simulator, Portals' API is made available to the user's code, to make the simulation of existing Portals code as easy as possible. So far we have used it to build online simulators, even though in the future this work could be extended to replay traces of instrumented Portals applications. In this section we will present the global architecture of the simulation, how we used the framework SimGrid to represent real-world behaviour, and how original Portals code is integrated in a simulation using our model.

A. Global architecture

Because modern interconnects can sustain bandwidths of several gigabytes per second, network transfers can have a latency of the same order of magnitude as PCI transfers between CPUs (or RAM) and NICs. This is why each node in the cluster is modeled as two distinct pieces of simulated hardware (which are called *Hosts* in SimGrid's vocabulary), which represent the CPU (with the RAM) and the NIC, connected by a PCI express link. This allows to model PCI transfers as well as communications between nodes (on the interconnect), and in the future it could also allow to model more precisely the processing time of message on the NIC independently from the computations on the main host. On these two hosts, four types of cooperative actors are created: the *"application logic"* actors on the *compute* host, and the *TX*, *RX* and *E2E* (which stands for End-to-End reliability) actors on the *NIC* host. This configuration of actors is represented in Figure 1, in which the *"User code"* is unmodified Portals code given by the user of our simulator (in the future we plan to support MPI code using OpenMPI and its Portals 4 byte transfer layer), the topology of switches connecting the machines in the cluster is specified in the platform's configuration file using SimGrid's built-in routers, and the other layers (Portals API and NIC host's actors) are programmed in our simulator using the SimGrid framework.

On the *compute* host, which models the main CPU and RAM of the compute node, *"application logic"* actors represent the behaviour the current application. These actors will run the user's code, to which they provide the Portals API as if the app was running on an actual system, and the

API calls will be handled by the simulator. It is possible to create several actors of this type on a node, to model several threads running on the same hardware.

On the second host, which models the NIC, three types of actors represent the behaviour of the NIC:

TX actors represent the transmission logic of the NIC, by receiving commands created by application logic actors through Portals calls and processing them. This processing involves making any Direct Memory Access (DMA) to fetch data from memory if necessary, sending a copy of the message to the E2E actor if it is enabled on this node, generating Portals events if necessary, and finally transmitting the message on the interconnect.

RX actors model the reception logic of the NIC, to process incoming messages from the interconnect. Depending on the type of message, RX will create an ACK or a complete response (with a payload) and pass it to TX actors to send it back on the interconnect. It also issues a DMA to write data to memory if needed, and generates any event that is required by Portals' specification.

Finally the E2E actor is responsible for the retransmission logic of lost messages, and its full behaviour is detailed in section III-D.

B. Mapping with SimGrid's mailboxes

In Simgrid, the prefered way to exchange messages between actors is through *Mailboxes*, which act as *rendezvous* points between actors: each actor can either put messages or get them from a mailbox, which will trigger a transfer in SimGrid's model. Mailboxes in SimGrid are not inherently linked with a specific host of the simulated platform, but by defining some actors as *permanent receivers* for specific mailboxes we can give the illusion that mailboxes are queues located on a specific host, which makes reasoning with them easier. In practice, this means that any communication issued to a mailbox will flow directly towards the receiver actor, instead of waiting for a corresponding *"receive"* on the mailbox. This allows commands to be transferred from *"application logic"* actors to the NIC even if the receiver on the NIC is not ready to "consume" the commands yet (for example).

In the proposed model, there are four types of mailboxes: on the *compute* host, a mailbox is bound to each Portals Event Queue (EQ), which makes it easy to send events from any actor on the NIC up the PCIe link. The three remaining mailboxes are on the NIC, and are used to send commands to each type of actor: one of them receives commands to be processed by TX actors, another one receives incomming messages from the BXI network (to be processed by RX actors), and the last one receives a copy of each message before it is transmitted on the network (so that the E2E actor can process it, and retransmit it if necessary).

The interaction between actors and mailboxes is depicted on Figure 2.

C. Modeling parallel processing of messages

In an actual NIC, most of the time several messages can be processed in parallel or at least in a pipelined way. At the lowest level, messages are usually split into packets of various size. For example, on the BXI NIC, there are circuits for DMA processing which have their own packet size for PCIe communications, and other circuits for transmission on the BXI link with a different packet size. The complete processing of a message involves going through each of these circuits, by using command queues at each step, which enables a pipelined progression of several messages. This low-level processing is designed this way because of the physical constraints of hardware components, but it is very complex, and modeling it entirely would make the simulations extremely slow. It would also make the model very tied to a specific NIC and take a lot of time to develop, for an improvement in accuracy that might not be necessary for our needs.

Since our model doesn't have the same constraints as the actual hardware, we use a simpler approach in order to model parallel or pipelined processing: RX and TX actors are responsible for the whole processing of a message, and we instanciate both of them several times on each NIC instead of modeling every hardware component and their interactions (which enable a pipelined processing of the messages in the actual hardware). We don't need to duplicate mailboxes, since several actors can fetch commands from a single mailbox. Although this is not as precise as a complete emulation of the hardware, it gives similar overall timing when measuring speed over several messages, while making the simulation faster.

D. End-to-End (E2E) reliability modeling

To ensure the quality of service and reliable communications, most modern interconnects use a retransmission system, which allows messages to eventually get to their destination even in the event of hardware failure on the interconnect. On the BXI NIC this mechanism is a dedicated hardware component, which has access to a small memory to store the contexts (also called *"E2E entries"*) which will keep track of the state of ongoing communications.

In our simulator the retransmission logic is modeled using a dedicated actor on each NIC host, which follows this algorithm: each time a TX actor sends a message across the network, a copy of the message (only the metadata, not the whole user payload) is sent to the E2E actor using the dedicated mailbox. This actor then waits until the delay for retransmission (timeout) is reached, and checks if we got an ACK for the considered message yet or not. If no ACK was received, a retry counter is incremented in the message structure, and it is passed to a TX actor to be retransmitted. If the message was acknowledged in time, the E2E actor discards it and moves on to the next one (wait until the retransmission date is reached, etc.).

On most interconnects, the timeout before retransmitting a message is constant throughout the whole application execution. This is an important property, because it allows us to have only one E2E actor on each NIC (instead of creating a

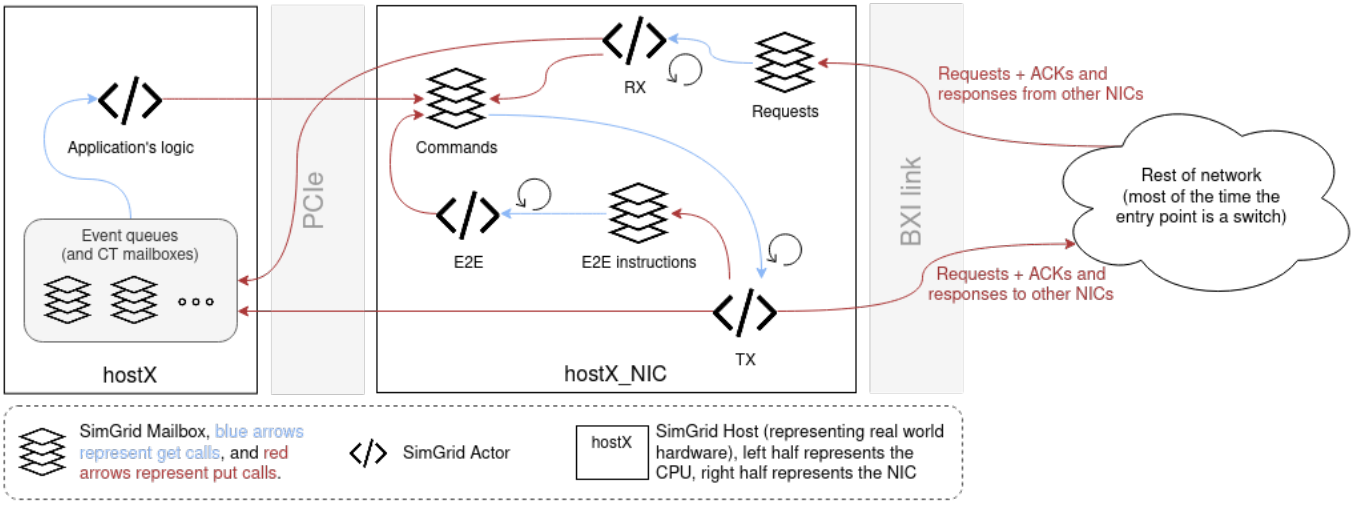

Fig. 2. Complete architecture with actors and mailboxes

new one for each message): indeed, because of this property, messages will always need to be retransmitted in the same order as they were created, which means a single E2E actor can process every message sequentially without missing any potential retransmission.

One of the difficulties in this E2E system is that Portals' ACKs are important messages which also need to be transfered in a reliable way, which is why the BXI interconnect has lower level ACKs to confirm the successful transfer of a Portals' ACK (or a GET response). This mechanism is fully modeled in our simulator and can be configured using either global or node-specific parameters, which allows users to get different tradeoffs between precision and speed, or explore what could happen for different E2E behaviours.

E. Exposing Portals API to simulated code

One of the main goals of our simulator is to allow the simulation of existing Portals code, with as few modifications to the original sources as possible. Unlike SMPI [11], which provide tools to compile existing MPI code with SimGrid in a single command, we do not yet have a compiler to make the process of simulating existing Portals application completely automated, but a simulation can easily be created from the original code using a simple C++ build configuration. This is made possible because the simulator exposes the Portals API (as specified in [1]), implemented in C++, in which function calls trigger operations in our model instead of writing commands to a real NIC. In order to achieve this, we created wrapper classes for most of the Portals C structures (network interfaces, event queues, portals tables, list entries, etc.), in order to keep the metadata needed by the model and have each structure's logic (mostly creation and destruction) well isolated.

Integrating the user code in our actor model is challenging: in an execution on a real cluster, each process is well isolated from the others, thanks to the operating system and the distributed hardware. This is not as straight-forward in simulation: every actor (and SimGrid's simulation kernel) runs in the same Linux process, and even though SimGrid natively has a very efficient mechanism to switch between contexts when waking up an actor [3], it is not sufficient to isolate the global variables of the user's code between *"application logic"* actors (which represent the different processes in the execution on a real cluster).

We considered several approaches to handle this difficulty: a naive solution is to make a specific *"application logic"* actors' implementation for each application. This means importing the user's code by transforming the application's functions into member functions of the actor class, and global variables into attributes of the class. This process works well because transforming global variables into attributes makes them private for each actor, as they would be if running the application on a real system, but it is not an easy process to automate, because it requires building a robust source-to-source tool, and we still have a problem if the user's code depends on an external library that uses global variables (such as OpenMPI [18] for example). Finally, it requires the user code to be valid C++ (some C code might not necessarily be compatible).

The approach we propose to solve these issues is similar to how SMPI [11] exposes the MPI API to its simulation actors: the user's code is compiled as a shared library, separately from the simulator (which allows the use of C code which would not be C++ compatible). It is then loaded by a generic *"application logic"* actor using the *dlopen* and *dlsym* functions (which allow us to open the library, fetch the *main* symbol and run it). The problem of making global variables and external libraries private to each actor can be solved by loading the user code's library and any external library several times with different names (so that the linker thinks they are different), but this is already well explained by the SMPI authors in [11], so we

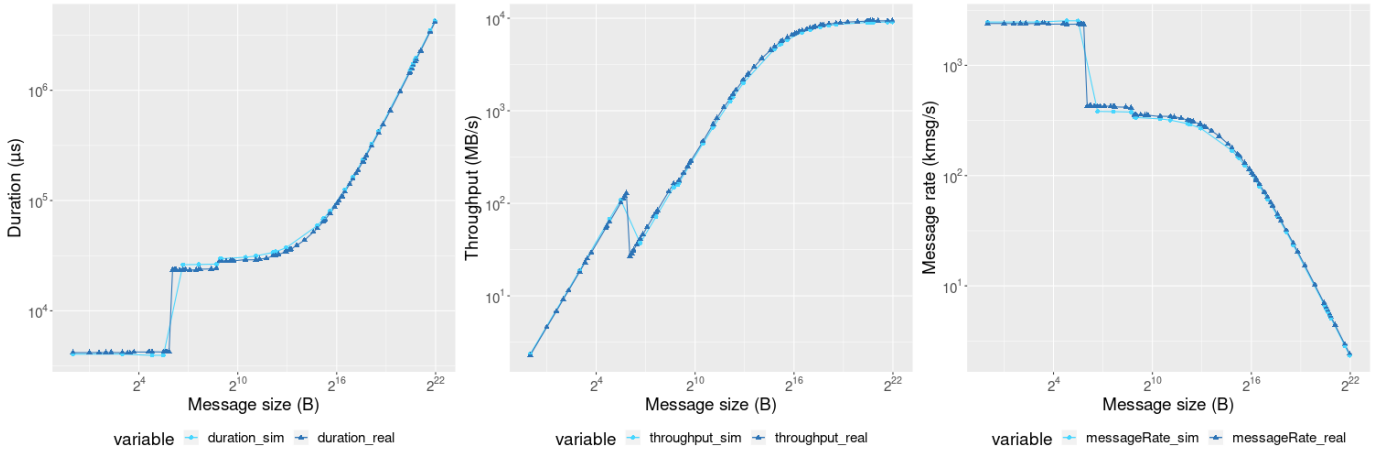

Fig. 3. Performance of Portals' PtlPut operation

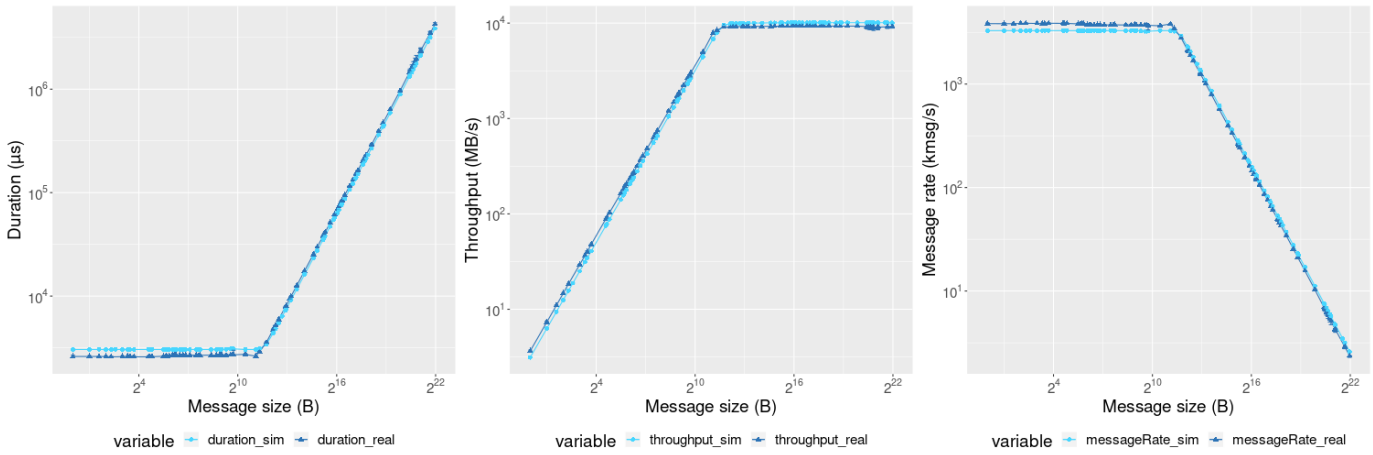

Fig. 4. Performance of Portals' PtlGet operation

won't go into any more detail in this paper.

In addition to Portals API calls, we implemented our own version of functions which manipulate time, such as *gettimeofday*, *sleep*, *nanosleep*, etc. Our implementation uses SimGrid Engine's simulated time, which enables users to use the exact same code to measure the performance of code on a real cluster or in simulation for example.

F. Options of the simulation

To allow faster simulations, the model has a few options, most of which can be configured globally using environment variables or locally for each actor in the XML deployment file (which specifies which actor gets executed on which host of the simulated platform): it is possible to skip the actual copy of the payload between actors when messages are exchanged, which saves time if most messages are big and the actual data is not important for the simulation (i.e. it doesn't change the execution flow). It is also possible to simplify the model to make its execution faster, by not modeling some PCI transfers (mainly commands from the compute node to the NIC, which are very small and sometimes have little impact on the results).

IV. Experimental validation

Validation experiments aim at comparing simulation results with real-world experiment on a BXI cluster. The nodes on this cluster use Intel® Xeon Phi™ Processor 7250 with a single BXI V2 NIC on each node. All the nodes used on the experiments are connected to the same BXI switch. The experiments that we present are point-to-point (between two nodes only), but we plan to design new ones to test the collective behaviour of a big number of nodes (using combinations of point-to-point operations since Portals doesn't have native collective operations, unlike MPI for example).

A. Portals' primitives

First we tested the model with the two basic Portals operations: *PtlPut* and *PtlGet* (results are shown respectively

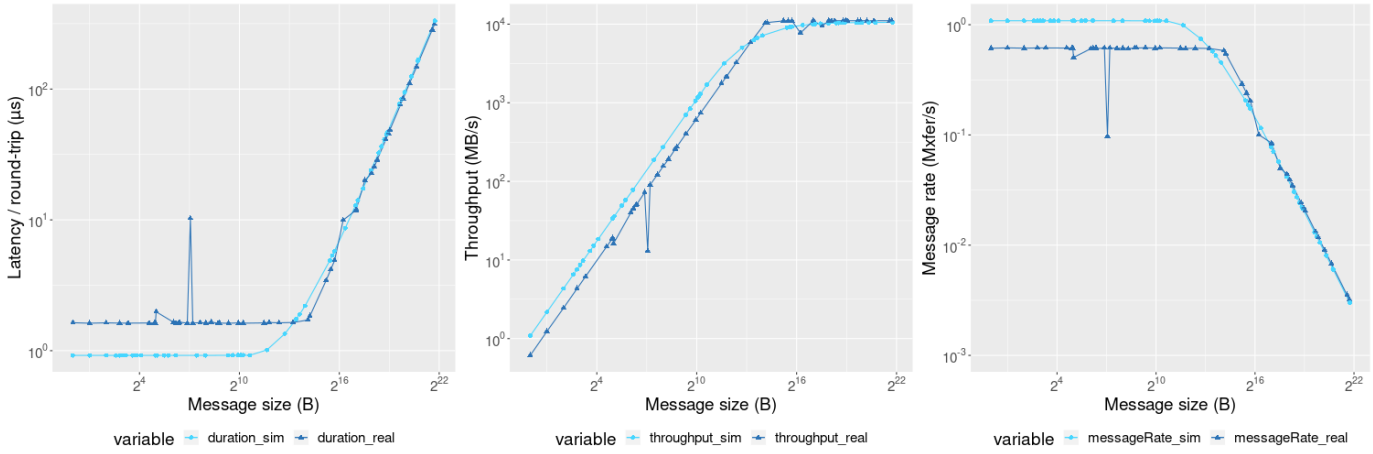

Fig. 5. Performance of ptlperf in match mode

on Figure 3 and 4). To have results as precise as possible we measured 10,000 operations (Put or Get), for 100 random sizes (between 1B and 4MB). For the real-life experiment each size was measured five times, but only one time in simulation: it is not relevant to repeat the same measurement since the simulation is deterministic.

PtlPut operations are more complex than *PtlGet*: because the payload is sent by the initiator of the request, the NIC uses several optimizations depending on message size to improve performance (such as caching the payload to reuse the user buffer faster, or sending the payload directly in the request to avoid a DMA from the NIC). These optimizations are specific to BXI (i.e. not mandatory according to the Portals specification), therefore modeling a different Portals NIC would require to modify this code, but we designed these specificities to be as separated as possible from the main Portals semantics' implementation.

On the other hand, with *PtlGet* requests, the NIC's algorithm is always the same: the initiator sends a header-only request, and the target always has to issue a DMA through its PCIe bus to fetch the payload and send the response.

In both cases we can see that the model gives good results even though we chose not to go for a full emulation of the NIC. There is still some room for improvement, but we believe that it can be done by adjusting the numerical parameters of the platform (such as PCIe latency for example), without any major model refactoring.

B. Simulation of real Portals code

We then tried to model more realistic code, taken from a tool used in production and written with Portals: ptlperf. It is used to measure the performance of a BXI cluster, and has various modes: *"as fast as possible"*, or *"realistic workload"*, etc.

As explained in Section III-E, we imported the original code of ptlperf in an *"application logic"* actor of the simulator, and tested it for 100 randomly chosen messages sizes between 1B and 4MB. Unlike the first tests, ptlperf doesn't send a fixed amount of messages while measuring the time, it sends as many messages as possible in five seconds while counting them. We used the *"match"* mode of ptlperf, which means that it will imitate the way a real application would send messages, instead of blindly sending as many as possible without waiting for completion events (which would be the *"fast"* mode). The results are displayed on Figure 5.

We can see that the performance estimation made by the simulation is not as good as for our first tests: in particular, the model is too optimistic for small messages. Our hypothesis is that ptlperf is a more complex tool than the previous experiments, and therefore that it spends a significant amount of time in CPU operations, which explains why the smallest messages always have the same duration regardless of their size: CPU is the bottleneck. Since the simulator doesn't have a model for compute time yet, the difference we see seems normal, and the bigger the messages get, the better the prediction is.

C. Performance of the simulator

For all the experiments the simulation was executed on a laptop with an Intel® Core™ i5-7300HQ and 8 GB of RAM, with every actor and the simulation kernel running in a single Linux process. In each case the simulation was executed twice, to see the influence of its configuration on the performance: the first time with the complete model, and the second time with the E2E processing disabled (in these simple examples we know there aren't any retransmission of messages so it doesn't affect the prediction). We recorded the execution time of the *PtlPut* and *PtlGet* experiments on the real cluster and on the simulator (as a reminder each test sends a million

messages: 10,000 for each message size with 100 different message sizes), which gives the following results:

- The *PtlPut* test took 35 seconds on the BXI cluster, and in simulation it took 39 seconds of wall-clock-time with E2E disabled and 209 seconds with E2E enabled.
- The *PtlGet* test took 34 seconds on the BXI cluster, and in simulation it took 87 seconds of wall-clock-time with E2E disabled and 276 seconds with E2E enabled.

As we can see, the E2E model is very costly (enabling it causes a factor of approximately 5 in execution time in the *PtlPut* test). Since the logic of this actor is very simple and shouldn't use the simulation kernel in an intensive way, we don't have any satisfying explanation, and optimizing this processing will be our next objective.

We also observe that simulating *PtlGet* operations is slower than *PtlPut* ones, although it takes the same amount of time to run on the BXI cluster. We suspect that our *PtlGet* code allows more requests to be in flight on the network at the same time, which doesn't matter in the real-life experiment (the performance is only bounded by the latency of the link for small messages, and the bandwidth for bigger ones), but could stress the simulation kernel excessively.

Finally, none of the simulations manage to be as fast as the execution on the BXI cluster (even though some are of the same order of magnitude in term of execution time). It is not alarming since these experiments are close to a worst-case scenario for the model: they are composed exclusively of network transfers, with no computation that we could skip on the simulation (whereas they would be executed on the real-life cluster). Nonetheless, scalability is going to be a challenge when modeling several (hundreds) of machines at the same time, and we plan to address this problem by allowing each machine to be modeled with a different level of precision: this will allow the simulator to simplify the model for the nodes of the cluster that are not subject to a heavy workload, therefore making the simulation faster while losing little in accuracy.

V. Conclusion

In this paper, we presented a new simulator, built on top of SimGrid to model the Portals 4 protocol used in HPC. It provides a standard Portals interface, and enables the simulation of unmodified application code, which allows developpers to run and optimize scientific applications in simulation, and to determine the influence of the interconnect's configuration on the performance of their code. We tested it for different types of operations, using both specific tests and existing Portals code, which demonstrated that the model gives good results for point-to-point operations, even though a CPU model could improve these results.

In the future we plan to extend the validation tests to collective operations, and also allow the simulation of MPI code, since few applications are written using native Portals directly.

References

[1] B. Barrett, R. Brightwell, R. E. Grant, K. S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson, "Portals 4.1 network programming interface," Sandia National Laboratories, Tech. Rep., 2014.

[2] S. Derradji, T. Palfer-Sollier, J. P. Panziera, A. Poudes, and F. Wellenreiter, "The BXI Interconnect Architecture," *Proceedings - 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, HOTI 2015*, pp. 18–25, 2015.

[3] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014. [Online]. Available: http://hal.inria.fr/hal-01017319

[4] "Infiniband Architecture Specification Volume 1 Release 1.4," Tech. Rep., 2020. [Online]. Available: https://cw.infinibandta.org/document/dl/8567

[5] "Infiniband Architecture Specification Volume 2 Release 1.4," Tech. Rep., 2020. [Online]. Available: https://cw.infinibandta.org/document/dl/8566

[6] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray ® XC™ Series Network Cost-Effective, High-Bandwidth Networks," 2012. [Online]. Available: https://www.alcf.anl.gov/files/CrayXCNetwork.pdf

[7] R. Brightwell, T. Hudson, K. Pedretti, R. Riesen, and K. D. Underwood, "Implementation and performance of Portals 3.3 on the Cray XT3," *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2005.

[8] G. Maglione-Mathey, P. Yebenes, J. Escudero-Sahuquillo, P. J. Garcia, and F. J. Quiles, "Combining OpenFabrics Software and Simulation Tools for Modeling InfiniBand-Based Interconnection Networks," *Proceedings - 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era, HiPINEB 2016*, pp. 55–58, 2016.

[9] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns, "Enabling Parallel Simulation of Large-Scale HPC Network Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 87–100, 2017.

[10] (2020) Top500 list - june 2020. [Online]. Available: https://www.top500.org/lists/top500/list/2020/06/

[11] A. Degomme, A. Legrand, G. S. Markomanolis, M. Quinson, M. Stillwell, and F. Suter, "Simulating MPI Applications: The SMPI Approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2387–2400, 2017.

[12] G. F. Riley and T. R. Henderson, "The ns–3 Network Simulator," *Modeling and Tools for Network Simulation*, pp. 15–34, 2010.

[13] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," 2008. [Online]. Available: https://doc.omnetpp.org/workshop2008/omnetpp40-paper.pdf

[14] "SST/macro 10.0: User's Manual," 2020. [Online]. Available: https://raw.githubusercontent.com/sstsimulator/sst-macro/v10.0.0_beta/manual-sstmacro-10.0.pdf

[15] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim - Simulating large-scale applications in the LogGOPS model," *HPDC 2010 - Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 597–604, 2010.

[16] E. G. Gran and S.-A. Reinemo, "InfiniBand Congestion Control, Modelling and validation," vol. 1, no. 2, pp. 390–397, 2012. [Online]. Available: https://www.simula.no/sites/default/files/publications/Simula.simula.362.pdf

[17] J. Cano, G. T. Fernández, F. J. Alfaro, and J. L. Sánchez, "OpaSim: an OPA Simulator for High-Performance Interconnections," Tech. Rep., 2018. [Online]. Available: https://www.dsi.uclm.es/descargas/technicalreports/DIAB-18-12-1/TechnicalReport.pdf

[18] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3241, pp. 97–104, 2004.