Programming Reconfigurable Heterogeneous Computing Clusters Using MPI With Transpilation

Burkhard Ringlein*†, Francois Abel*, Alexander Ditter†, Beat Weiss*, Christoph Hagleitner*, and Dietmar Fey†
*IBM Research Europe, †Friedrich-Alexander University Erlangen-Nürnberg
{ngl, fab, wei, hle}@zurich.ibm.com, {burkhard.ringlein, alexander.ditter, dietmar.fey}@fau.de

***Abstract*—With the slowdown of Moore's law and the stop of Dennard scaling, energy efficiency of compute hardware translates to compute power. Therefore, High-Performance Computing (HPC) systems tend to rely more and more on accelerators such as Field-Programmable Gate Arrays (FPGAs) to fuel high demanding workloads, like Big Data applications or Deep Neuronal Networks. These FPGAs are reconfigurable and sometimes no longer bus-attached to a CPU but directly connected to the data center network fabric as standalone nodes. This mix of CPUs and FPGAs leads to the creation of Reconfigurable Heterogeneous HPC (ReH^{2}PC) clusters for which no established programming model exists, despite many proposals in the past.**

In contrast to this, the Message Passing Interface (MPI) has evolved as the de-facto standard to program classical HPC clusters, due to its high-re-usability and fast development of applications. This paper revisits the programming model of ReH^{2}PC clusters and argues that MPI is suitable for programming heterogeneous clusters of FPGAs and CPUs.

We demonstrate a one-click solution for compiling and deploying a standard MPI application on ReH^{2}PC clusters. Our framework implements a High-Level Synthesis (HLS) library, a specific run-time environment for FPGAs and CPUs, and a transpiler that closes the semantic gap between the MPI API and FPGA designs.

Our experiments with 31 FPGAs show an average speedup of 4 and a 90% reduction of power consumption compared to a cluster of CPUs.

***Index Terms*—MPI, network-attached FPGA, stand-alone FPGA, transpilation, partial reconfiguration, data centers, heterogeneous programming model, heterogeneous clusters**

I. Introduction

The end of Dennard scaling and the wind down of Moore's law boosted heterogeneous architectures in all areas of computing. Therefore, Data Centers (DCs) have been equipped with specialized Co-processors, GPUs and more recently, with Field-Programmable Gate Arrays (FPGAs) to increase compute power while keeping the energy densities in manageable ranges [1]–[3]. In addition, the demand for High Performance Computing (HPC) services is increasing further. Today, the largest supercomputers are heterogeneous (using GPU acceleration) and there is a lot ongoing research to integrate reconfigurable devices into Reconfigurable Heterogeneous HPC (ReH^{2}PC) systems.

To be of practical use, heterogeneous FPGA+CPU systems must come with an efficient, flexible and — if possible — easy programming model to allow users to orchestrate their algorithms. Hence, there have been multiple approaches to develop distributed FPGA platforms since the mid 2000s (among others [4]–[16]). However, none of these platforms has been widely adopted and today the largest HPC systems with FPGAs are still limited to a few tens of nodes [16].

This situation is in stark contrast to the classical HPC world, where one framework has been established as the de-facto standard: the Message Passing Interface (MPI). The advantage of settling on one established standard is the acceleration of research and the development of better applications. In the classical HPC world, MPI is the foundation for most simulations, large big data stacks, and even machine learning.

In the meantime, we noticed that FPGAs have started to become directly attached to the DC network and that some of them are even starting to operate as standalone nodes [1], [3]. This is a profound change of paradigm in CPU–FPGA and FPGA–FPGA interactions. It opens new perspectives on the deployment, scale-out, portability and re-usability of FPGAs in heterogeneous HPC applications.

These observations motivated us to reconsider the use of MPI for ReH^{2}PC platforms. The use of MPI could reduce the semantic gap between HPC applications and reconfigurable hardware (HW), because software (SW) engineers know how to use MPI. A common standard would also guarantee the same behavior of the MPI Application Programming Interface (API), whether executed on a CPU or an FPGA. Finally, ReH^{2}PC systems could benefit from the lessons learned during two decades of MPI evolution and optimization.

On the other hand, not all design paradigms of MPI are also suitable for ReH^{2}PC. For example, the principle of buffers — "compute first, send later" — does not fit the inherent parallel processing capabilities of FPGAs. In the same sense, the "Single Program, Multiple Data" (SPMD) notion may lead to wasting of FPGA logic. Hence, in order to use the standardized MPI language to program FPGA clusters, it must be trans-compiled, or *transpiled*.

Besides the points listed above, this work argues for the use of MPI as programming model for ReH^{2}PC because it is a perfect starting point for a source-to-source compilation towards modern High-Level Synthesis (HLS) tools for FPGAs. Therefore, we contribute a proof of concept MPI implementation on ReH^{2}PC clusters consisting of a transpiler, FPGA and SW run-time modules, an HLS synthesizable MPI library and a program for the automatic deployment and execution of the application.

The rest of this paper is structured as follows: The next section revisits the programming of ReH^{2}PC and argues in detail for the use of MPI. We then discuss related work before

This is the accepted version of the article published by IEEE: DOI 10.1109/H2RC51942.2020.00006

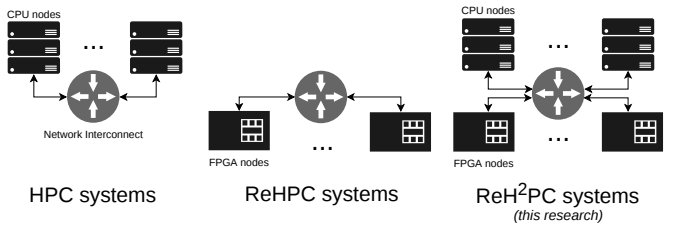

Fig. 1. HPC vs. ReHPC vs. ReH^{2}PC (based on [17] and [18])

we describe our implementation to realize such a system. Finally, we evaluate our framework with a 2d-stencil app.

II. Programming ReH^{2}PC Clusters

Today's high-performance computing systems can be classified into three classes as depicted in Figure 1. The first and traditional HPC class solely consists of CPU nodes, while the second class, typically referred to as Reconfigurable HPC (ReHPC), is only composed of FPGA nodes. Such an FPGA node can either be i) a classical PCIe bus-attached FPGA that acts as a co-processor slave under the control of a CPU or ii) a network-attached FPGA with direct access to the cluster interconnect [2]. The third class comprises a mixture of the CPU servers from the first class and FPGA nodes from the second class. Consequently, it is named ReH^{2}PC [18].

Today, the number of heterogeneous clusters is increasing rapidly as many large-scale parallel applications can benefit from the incorporation of FPGA hardware accelerators.

Unfortunately, despite many attempts, no standard has yet emerged for programming such heterogeneous clusters. This absence of an agreement hinders the rapid development of applications using FPGAs in HPC. Therefore, a programming model to enable the development, deployment and management of mixed clusters of CPUs and FPGAs is highly desirable.

Among the FPGAs used in ReHPC and ReH^{2}PC, network-attached FPGAs are attractive because they offer a higher scalability potential compared to classical bus-attached ones [1] [3]. Such FPGAs directly attach to the cluster interconnect and can communicate over 10/100Gb links with standard protocols such as IP/TCP/UDP and Infiniband. Furthermore, the availability of a direct and standardized communication path to the FPGA — without going over the PCIe bus or the CPU first — opens new programming and clustering possibilities for heterogeneous clusters of such FPGAs.

These observations motivated us to reconsider the use of MPI for ReH^{2}PC platforms. Making MPI available as a programming model for the ReH^{2}PC domain would speed up the development cycle of new HW clusters on one hand and new applications on the other, since both sides would benefit from an established interface.

In the remainder of this section we revisit the programming model for ReH^{2}PC and argue for the use of MPI. We first give a very brief introduction to MPI as used in HPC systems before justifying our effort to enable MPI for network-attached FPGAs. Afterwards, we introduce our transpilation tool and provide the details of the run-time modules implementation.

```
void MPI_Send(void* data, int count, MPI_Datatype datatype, int destination,
       int tag, MPI_Comm communicator);

void MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag,
       MPI_Comm communicator, MPI_Status* status);

void MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root,
       MPI_Comm communicator);

void MPI_Scatter(void* send_data, int send_count, MPI_Datatype send_datatype,
       void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root,
       MPI_Comm communicator);

void MPI_Gather(void* send_data, int send_count, MPI_Datatype send_datatype,
       void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root,
       MPI_Comm communicator);

void MPI_Reduce(void* send_data, void* recv_data, int count, MPI_Datatype datatype,
       MPI_Op op, int root, MPI_Comm communicator);
```

Listing 1. Signatures of popular MPI calls.

A. The Message Passing Interface for CPU clusters

Over the last two decades, MPI has become the dominant programming method used in HPC clusters. Its portability and efficiency attracted a large base of users who embraced it as de facto standard for developing an abundant number of scientific applications.

In MPI vocabulary, a program that runs on a node is called a *process*. A *rank* is a specific integer number assigned to such a process during its initialization in order to identify that process in a parallel multi-processing program. Processes communicate with each other using a concept of message passing. These messages are packets of data encapsulated into *envelopes* that contain routing information. The transfer of data is called a *send* and the receiving of data by a process is called a *receive*.

The basic communication calls — and the calls to which all blocking MPI routines can be reduced — are `MPI_Send` and `MPI_Recv`. A data transfer involving these two routines is said to be synchronized because data is only transmitted when both sender and receiver are ready. This synchronization is established by the MPI run-time environments of the concurrent processes via a handshake process. The method signatures of `MPI_Send` and `MPI_Recv` are shown in Listing 1.

The most powerful feature of MPI are the defined collective routines. For example, with the single command `MPI_Bcast` data are broadcasted from one rank — `int root` – to all others. If an application does not need the same data at all nodes, `MPI_Scatter` can be used to distribute different portions of the data to different nodes. At the end of a computation, `MPI_Gather` can combine them again into one buffer using a single line of code. Alternatively, different data can also be *reduced*, where the data from all nodes is combined using a reducing function like `MPI_SUM`. The mentioned routines are also presented in Listing 1.

Originally, MPI followed the SPMD paradigm. However, different processes can be assigned to execute different parts of the program based on their rank number. This SPMD assumption simplifies the deployment of the program for users and for the run-time environments. However, it only works well if the cluster solely consists of homogeneous computation nodes.

In the presence of heterogeneous nodes, the user may want to tell the run-time environment which part of the program runs better on which hardware. Here, *Heterogeneity* is meant

in terms of available cores, processor architecture, clock speed and memory capacity, but may also include the presence of an accelerator such as a GPU or an FPGA [19], [20]. To address this issue, MPI provides the concept of *affinity* [21], [22]. The notion of affinity allows a programmer to define which rank should be scheduled on which host or group of hosts before the program starts. As a consequence, the concept of affinity weakens the SPMD paradigm of MPI but offers possibilities to efficiently use heterogeneous clusters.

To not exceed the scope of this paper, we refer the reader to standard references like [23] or [24] for more details on MPI.

B. Message Passing Interface for CPU+FPGA clusters

We want to enable HPC application developers to leverage FPGAs as seamlessly as CPUs. This implies the execution of existing HPC applications on ReH^{2}PC clusters without any code modification. Therefore, we propose to use a well-known programming model and re-adjust it for FPGAs at compiler-level. To achieve this goal, we use MPI with transpilation to adapt the HPC application to the particular features of a ReH^{2}PC cluster.

In general, there are two ways of introducing a programming model to ReH^{2}PC. The first is to invent something completely new, which is then optimized for this domain from the beginning. The second way is to build on well known and established concepts from older technologies and adapt them to the new environment. This research will argue for the latter case. Given the wide-spread adoption of MPI, we want to use its standardized syntax and semantics as a single programming model for ReH^{2}PC clusters.

Programming languages offer two ways to synchronize concurrent computations: The nodes synchronize either implicitly via a shared memory address space, like e.g. OpenCL, or explicitly via the exchange of messages via a shared network, like e.g. MPI. Furthermore, a programming model for large scale ReH^{2}PC clusters should be compatible with default OSI models of communication and therefore should be independent of the used communication topology and protocols. Since network-attached FPGAs are distributed memory systems interconnected with standard OSI protocols, the modeling of concurrent programs with messages fits their architecture better than the implicit synchronization with memory. In addition, the Register-Transfer Level implementation of a network protocol stack typically provides a one order of magnitude lower latency than a software counterpart. As an example, the TCP stack of the network-attached FPGAs in [25] comes with a latency of $2.8\ \mu s$ which compares favorably to the average ~$100\ \mu s$ of an SW stack. This low latency network interface renders the use of extra accelerators for MPI communication and collective operations, like introduced by [26] or [27], unnecessary.

Next, the parallel processing capabilities of FPGAs have the potential to overcome the performance bottleneck related to the sequential "compute first, communicate later" pattern of MPI models on CPUs [12]. This conversion of buffers to streams is possible during the transpilation step, as explained in the next section. Our proposed approach to optimize MPI programs for a specific hardware target at compile time is also a common practice in performance tuning of petascale and exascale HPC applications [28]. Of course, the consideration of run time properties at compile time weakens the SPMD paradigm, but as discussed in Section II-A the affinity concept of MPI has similar consequences. In addition, [19], [20], [28] pointed out that even the characteristics of similar CPUs, such as different frequencies or available memory, should be taken into account at compile time to leverage these clusters.

Finally, even in a pure stream-based environment like [12], the implementation of collectives is not possible without synchronizing messages between the run-time environments. Hence, the abstraction level of the MPI API calls fits not only the idea of distributed heterogeneous nodes but also the paradigm of HLS languages and allows the detection of buffers that should be turned into streams during compilation.

For all these reasons — the conform level of abstraction, the compatibility with TCP/UDP/IP, and the fact that it is an established standard in the HPC world — we decided to develop an MPI implementation for ReH^{2}PC clusters and not to invent a new standard that is inspired by MPI.

III. Transpilation of MPI — Or how to square the circle

MPI was designed for homogeneous CPU clusters and was later adapted to heterogeneous cluster environments employing concepts like affinity. We continue on this path by adapting CPU code to FPGA designs by exploiting MPI semantics at compilation time. Before we present the details of our proof of concept implementation of MPI, we explain the main ideas behind the transpilation of MPI.

A. Why transpilation can close the semantic gap

MPI is not only an API, but mainly a programming model that follows the Bulk Synchronous Parallel computation model. With MPI, the communication between processes and the computing- and communication-phases within these processes are clearly separated. Hence, a compiler knows the dependencies between each sequential thread of the program and the direction and timing of the dependencies. This situation is very similar to the way many FPGA designs are described by VHDL, Verilog, or HLS, i.e. a set of sequential processes and their directed interconnections. Consequently, the abstract concepts used to describe FPGAs and MPI clusters are dual.

To map MPI to its FPGA counterpart, some transformations are required that respect the special characteristics of ReH^{2}PC. The most dissimilar part is the strict sequencing of computation, storing the result in buffers and the transmission of these buffers. However, if the access pattern of an array is known, modern HLS tools already turn this knowledge into a parallel computation if possible. In order to perform this transformation, the HLS compiler needs to recognize important parameters like the size of the buffer, the operational window, and the incoming and outgoing bandwidth, i.e. how fast and at which interval the next or previous computation step

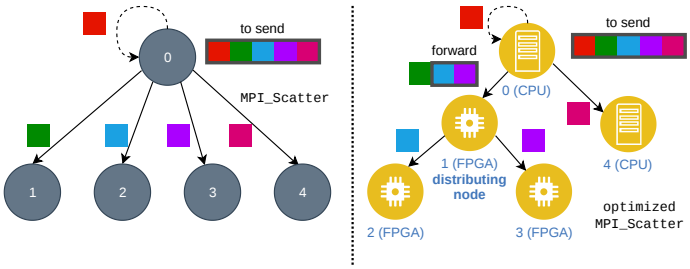

Fig. 2. Optimization of `MPI_Scatter` using a tree transformation and leveraging the lower network latency of FPGAs. (The first data chunk remains at the root node.)

```
{
  "nodes": {
    "cpu"  : [0, 33],
    "fpga" : "1 - 32"
  }
}
```

Listing 2. A cluster description in JSON.

```
void MPI_Send(
// ----- MPI_Interface -----
  stream<MPI_Interface> *soMPIif,
  stream<Axis<D> > *soMPI_data,
// ----- MPI Signature -----
  int* data, int count, MPI_Datatype datatype,
  int destination, int tag, MPI_Comm communicator);
```

Listing 3. Vivado HLS compatible signature for `MPI_Send`.

can process the data. Therefore, even the explicit semantic of the MPI API calls (like `MPI_Send()` and `MPI_Recv()`) can be converted to stream-like read and writes [29, p. 20].

Following this path, to map MPI to synthesizable HLS programs, a transpiler requires the description of the final cluster setup at run time. This description contains the number of CPUs and FPGAs that will be part of the computation and which ranks will be associated with which one of them. An example of such a description for a cluster with 2 CPUs and 32 FPGAs in JSON format is provided in Listing 2.

Next, another mismatching concept between MPI and FPGA clusters is the SPMD paradigm. In a pure software world, SPMD simplifies development and debugging of parallel programs. This comes with relative low costs at run time, because if a concrete rank only executes one part of the program, the unused part of the binary is not touched and only "wastes" space in memory.

However, this situation is very different for FPGAs, where unused logic also wastes logic at run time and therefore limits the overall resources that are available to solve the original task. Consequently, an MPI transpiler for ReH^{2}PC needs to split the original SPMD program into a "Multiple Program, Multiple Data" (MPMD) version to ensure that only used parts of a specific node are also synthesized to (expensive) FPGA logic and to avoid "dead code". This split into multiple rank-specific versions of the MPI program can also be done at transpilation time using the described cluster description.

Third, collective routines are a powerful abstraction but to execute them the MPI run-time environment needs to determine the exact execution of each instance during the operation of the program. For example, which node is the sender and which are the receivers in the case of a specific `MPI_Bcast`. By using the provided cluster description, these decisions can also be brought forward to compile time. In addition, at this point in the transpilation, the optimizations of the MPI collectives, e.g. with neighborhood communications [30] or hierarchical typologies [31], can be decided and prepared. Furthermore, these optimizations can leverage the advantages of ReH^{2}PC environments and can schedule the optimizations with respect to the properties of the final hardware node. For example, combining or distributing network messages is faster on FPGAs than on CPUs, and therefore the overall performance of collective routines can be increased by the transpilation. One example for such an optimization of `MPI_Scatter` is given in Figure 2.

In addition to all these adaption and optimization steps, one additional advantage of using a transpiler at this abstraction level is the possibility to generate detailed warnings or errors, because subsequent synthesis steps may miss some information to guide users in a helpful way.

Finally, all the above adaptations enable the transpilation tool to emit a specific FPGA code in the language of the targeted HLS tool. The details of our transpilation process are described in the following.

B. How to transpile

Our proof of concept implementation transpiles MPI C code to synthesizable HLS code for Xilinx Vivado. The number of supported languages is therefore limited to a single input and a single output language but it is sufficient to demonstrate the general concept of transpilation. Our transpilation tool is called `ZRLMPIcc` and it builds on top of our previous research [18].

First, we remove all parts of the code that will not run on the FPGA using a static code analysis investigating the description of the cluster at compile time. We feed the code to a C preprocessor[1] and then parse the MPI C code to an Abstract Syntax Tree (AST) using the python library `pycparser`[2]. The AST is a representation of the program in tree structure, which gives us a powerful tool to manipulate the program. Using the provided JSON description of the cluster and the generated AST, we set the rank and size variables, as if the program were to be executed, and then determine which part of the code will not be executed by the FPGA nodes. Alternatively, the user can guide this code split by manually annotating the code with `ZRLMPI_SW_ONLY` and `DEBUG` pragmas. These optimizations avoid the instantiation of "dead code" in the FPGA, which would waste FPGA resources. However, by using the AST representation, the transpiler can detect if the execution flow of a program is the same for some ranks. Therefore, we avoid generating an identical bitfile multiple times.

Second, constant values, such as the cluster size, are propagated through the AST to allow further optimizations by the HLS tools. In addition, a constant folding is performed.

Afterwards, still using the AST and the information about the rank associations, the transpiler analyzes the collective routines and substitutes the original code with the necessary steps that each rank has to execute. One example of such a substitution is given in Figure 3 for the `MPI_Scatter` routine. Since the explicit MPI semantic states all necessary

[1]GNU project C and C++ compiler (GCC), version 9.1.0

[2]pycparser: Complete C99 parser in pure Python, version 2.19: https://github.com/eliben/pycparser

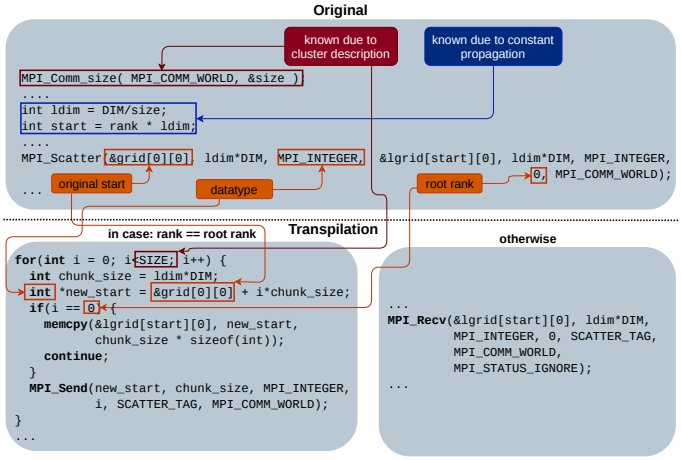

Fig. 3. Transpilation example of `MPI_Scatter`. All parameters of the replacement snippets are based on the original API call, but only some of them are highlighted as examples in the figure.

information, ZRLMPIcc knows the root rank for this routine, the datatype of the buffer, and the original start address. Combining this with the information about the cluster enables the transpiler to replace the collective routine with the equivalent `MPI_Send` and `MPI_Recv` calls for each rank. Based on the source and target buffers, a `memcpy` is also inserted for the root rank if necessary. During this step, optimizations of the collective routines can be prepared, e.g. as sketched on the right hand side in Figure 2. If this option of the transpiler is enabled, the topology of the optimized collective routines are automatically derived from the cluster description by an algorithm which decides which ranks become distributing nodes and which ranks receive from them.

Fourth, using the AST, ZRLMPIcc ensures that the generated C code for the FPGA is synthesizable, e.g. by renaming the `main()` function and passing the necessary HLS stream data structures to the MPI methods. This is necessary because the MPI API calls may be called multiple times by the application, but we have only one connection to the run-time environment in the hardware. Hence, the objects of the interface streams to the run-time environment must be present in the HLS program at all times. One signature of a modified MPI API call is listed in Listing 3. There are additional methods for the other API calls as well as versions with the data parameter of type `float`. From this rewritten AST we generate a C code file for the FPGA ranks. These generated C files may still require the information of the rank at run time, if the remaining part of the code does not differ between different FPGA nodes.

Afterwards, the generated code will be inserted into an HLS project and synthesized. We implemented the MPI API calls in an HLS synthesizable library, which also provides the necessary header file `mpi.h`. This library contains also a wrapper for the main function to control the start and stop of the application. These control signals are handled by the run-time environment to prevent the FPGA nodes from starting the execution before the complete cluster is ready.

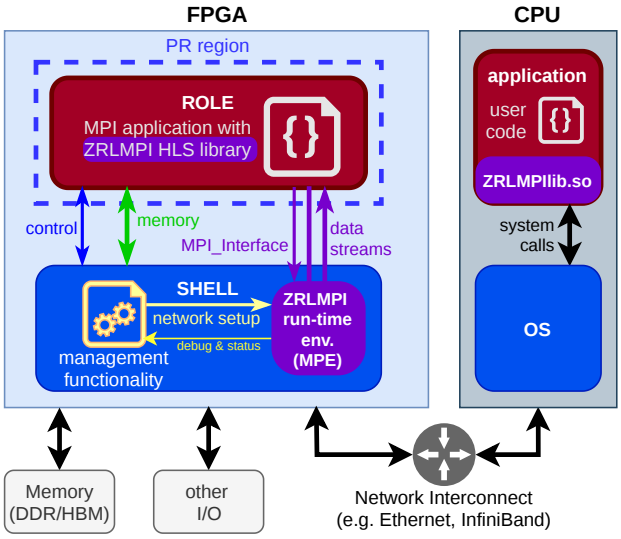

Fig. 4. The SRA of a ZRLMPI FPGA design and the equivalent components in a CPU. The MPI application is instantiated inside the Role and connected to the MPE core through one MPI_Interface and two AXI4-Streams. The MPE is connected to the network stack inside the Shell (not shown).

IV. Implementation

This section presents *ZRLMPI*, a proof of concept implementation for running MPI applications on heterogeneous clusters with FPGA and CPU nodes. This work extends the ZRLMPI framework first introduced in [18]. The overall architecture is depicted in Figure 4 and consists of a Shell and a Role (IV-A), a ZRLMPI run-time environment (IV-B and IV-D) that is connected with the transpiled application via the ZRLMPI interface (IV-C), and the CPU run-time environment.

Finally, we introduce our one-click solution in IV-E.

A. FPGA Architecture Schema

The Shell Role Architecture (SRA) is a design pattern that can be found in many recent FPGA designs. The idea is to separate the platform-specific parts from the application-specific parts in order to increase the re-usability of both platform and application. The first part is called the *Shell* and contains all necessary I/O components, the network stack, and all required run-time modules. The Shell is the conceptual counterpart of the Operating System (OS). The application-specific part of the logic is referred to as *Role* and corresponds to the CPU application in Figure 4. An SRA always requires a fixed interface between Shell and Role, otherwise the Shell would have to be modified for every application and vice versa. This Shell Role Interface (SRI) should be as generic as possible, for a single Shell to support a wide range of Roles.

The logic of the Role is typically controlled by the user and deployed using *Partial Reconfiguration*. On the other side, the logic of the Shell is controlled by the platform provider, to guarantee the integrity of the infrastructure. An example for an SRA can be found on the left-hand side of Figure 4.

B. ZRLMPI components and scope

The goal of ZRLMPI is to provide a transpiler for the user to translate her/his original MPI code, which runs with e.g. OpenMPI or MPICH, to a partial FPGA bitstream as well

as a SW binary for the CPU. Afterwards, the user can start his/her program simply by launching the ZRLMPI execution environment (`ZRLMPIrun`), which takes care of configuring deploying and executing the application on a cluster of CPUs and FPGAs.

In order for ZRLMPI to work as a platform and application independent programming framework, we implement a specific run-time module for the FPGAs and one for the CPUs, as depicted in purple in Figure 4. The former is called the `Message Passing Engine` (MPE), and its SW counterpart is named `ZRLMPIlib`. These run-time modules implement the ZRLMPI protocol that synchronizes all FPGA and CPU nodes as specified by the MPI API and connect the application to the FPGA communication interfaces or the OS. The FPGA run-time environment will be addressed in subsection IV-D. The CPU counterpart is written in C++ and linked as shared object to the user application at run time. To compile the application inside the FPGA and connect it with to MPE, a `ZRLMPI HLS library` was created that maps the API calls to streams. This HLS library is the implementation of the `"mpi.h"` header that is included by the application. It implements all necessary MPI APIs as shown in Listing 3 and maps these to the stream interfaces of the MPE, as depicted by the purple arrows in Figure 4. The FPGA and CPU run-time environments ensure the synchronous execution of the MPI API calls using handshakes. For the sake of this demonstrator, we only need to support the `MPI_Init`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Send`, `MPI_Recv`, `MPI_Scatter`, `MPI_Gather` and `MPI_Finalize` methods.

C. MPI interface in the FPGA

Apart from data streams, the Role application needs to send the meta data of the MPI API calls to the MPE. The type of this meta data stream is shown in Listing 4 and is also marked by a purple arrow in Figure 4. ZRLMPI aims at being portable to other FPGA environments while reducing the effort of adapting and deploying one or multiple MPI applications as Roles. Therefore the interfaces from and to the runtime environment are as generic as possible. In the network-attached case, the MPI run-time environment must also ensure that the user does not break out of his subnetwork or attack the DC network. Therefore we decided to make the FPGA run-time environment part of the Shell in our design.

Consequently, the meta data stream between app and MPI run-time module must be implemented at HDL level as part of the SRI. Since AXI4-Streams are commonly used as data paths inside FPGA designs, we decided to use them for implementing the MPI interface between the ZRLMPI run-time environment and the FPGA application. Next, to keep the SRI simple, there should not be a stream for every possible type of call, instead there should be one generic interface, which is mapped to the original MPI calls by a small library inside the Role. Finally, since every MPI call is always initiated by the application, the `MPI_Interface` stream is only an output for the Role. For example, the specification of `MPI_Recv`

```
type tMPI_Interface is record
    mpi_call : std_logic_vector( 7 downto 0);
    count    : std_logic_vector(31 downto 0);
    tag      : std_logic_vector(31 downto 0);
    rank     : std_logic_vector(31 downto 0);
end record tMPI_Interface;
```

Listing 4. VHDL description of the `MPI_Interface` type.

guarantees that only the data that fits the requested data type and came from the right source is returned.

D. FPGA run-time module

The ZRLMPI run-time environment implementation inside the FPGA is a Finite State Machine (FSM) that processes the messages of the application and maps them to the ZRLMPI protocol and the FPGA network stack. The MPE is connected to the Role through one MPI interface, as shown in Listing 4, and two data streams, as depicted in Figure 4. On the other side, the MPE is connected to the data path of the network stack (UDP or TCP). Only one MPI interface to the application is accurate, because according to the MPI specification, only one MPI task can be active at a time. In order to be compliant with the MPI specification, the MPE only starts acting upon a new `mpi_call` on the `MPI_Interface`.

Additionally, the MPE needs to map IP addresses and ports to the rank as specified by MPI and vice versa. Therefore, ZRLMPI provides a mapping table that is distributed to all nodes of the heterogeneous cluster during their deployment. For debugging, the MPE also provides internal status information, like how many packets have been processed, the internal state of the protocol engine, or a dump of the last processed packet, to the management functionality of the Shell, via an AXI4-Lite bus, as sketched in Figure 4. Finally, the MPE receives the node rank information and propagates this to the Role.

E. ZRLMPIrun: The one-click solution

The two key elements of ZRLMPI are the possibility to bring real MPI code to FPGAs and that the whole ReH^{2}PC cluster can be started with only one command, similar to other MPI implementations. The first key objective is achieved by ZRLMPIcc. For the second goal, we need an FPGA platform that allows the distribution of the partial bitfiles of the Role and the configuration of the network interfaces within the FPGA in an automated fashion. Also, we need to distribute the information about the individual rank and the cluster routing table to the FPGA bitfiles. We implemented a deployment framework in python that allows to distribute the partial bitfiles of the application to the FPGAs automatically and launches all CPU instances. This program is called ZRLMPIrun.

V. Evaluation

To evaluate our approach of using MPI with transpilation as programming model for ReH^{2}PC, we implemented a two-dimensional stencil application using ZRLMPI.

A. Experimental Setup

We use the FPGA platform introduced in [2] to run our experiments. This platform integrates 32 Kintex KU60 FPGAs[3]

[3] `xcku060-ffva1156-2-i`

TABLE I
JACOBI MPI APPLICATION EXECUTION TIMES AND POWER USAGE

Data size	cluster size	CPU only		ReH²PC (1 CPU, size-1 FPGAs)	
		ms / iteration	avg. **Watt**	**ms** / iteration	avg. **Watt**
16x16	2	23.33	254.49	7.59	136.24
256x256	4	1670.90	508.48	444.77	154.21
256x256	8	2136.72	1018.08	842.58	187.68
1024x1024	8	43120.16	1017.99	7763.87	187.09
1024x1024	16	36261.91	2036.16	9311.71	253.23
1024x1024	32	41021.51	4072.33	20939.65	387.99
256x256 (with tree optim.)	8	1686.55	1017.68	637.87	187.97

with one Intel FM6000 switch onto a passive water cooled carrier board. The switch acts as a leaf switch that aggregates 32 10GbE links from the FPGAs and connects them to the core of the DC network via $8 * 40$GbE up-links. Each FPGA can access 16GB of DDR4 RAM. The clock cycle of the FPGAs is 6.4 ns. For our benchmark we use only UDP.

Next, we need a management architecture that allows us to distribute the partial bitfiles and necessary run time information to the FPGAs in an automated fashion. The system architecture described by [3] allows the deployment of applications by sending the partial bitfile to the FPGA, as an HTTP `POST` call. We therefore use this framework as the basis for ZRLMPIrun.

For the CPU examples, we use bare metal servers with an Intel i7 CPU 960 @ 3.20GHz and RHEL 7.8 as OS.

B. Jacobi-2D example application

We implement the simple Jacobi iteration for approximating the solution of a two-dimensional Laplace equation (also called steady-state heat equation). A code snippet containing the main execution loop is given in Listing 5. This source code is compiled with ZRLMPIcc to run on both FPGAs and CPUs. The deployed ReH²PC clusters always consists of `size`-1 FPGAs and one CPU. We executed this example with different sizes on CPU only and ReH²PC clusters. The execution times and power measurements are shown in Table I. The measured execution times per iteration are averaged over five iterations and timed on the root node. The power is measured externally per node and accumulated for the used cluster. On average, a CPU node consumed 127 W and an FPGA node 8.5 W. To make the execution times comparable, we ensured that all cluster setups have similar `ping` times. We used the 1024x1024 application with cluster size 8 to evaluate the resource overhead of the FPGA MPI run-time environment in comparison to the FPGA app. This comparison is shown in Table II.

In a second step, we wanted to evaluate our proposed optimization of collectives in ReH²PC as discussed in Section III-A. The result is also presented in Table I. The topology automatically derived by ZRLMPIcc for the test case with tree-optimized collectives is depicted in Figure 5.

The result shows that a cluster that leverages FPGAs is always faster than a pure CPU setup and our ReH²PC approach outperforms the classical HPC by a factor of 4 on

TABLE II
RESOURCE USAGE OF ZRLMPI IN A XCKU060

Resource	Available	Used		
		Total	**MPE**	**APP**
LUT	331680	113832	1940	1715
LUTRAM	146880	11103	8	0
FF	663360	132553	1744	1058
BRAM	1080	609	3	320

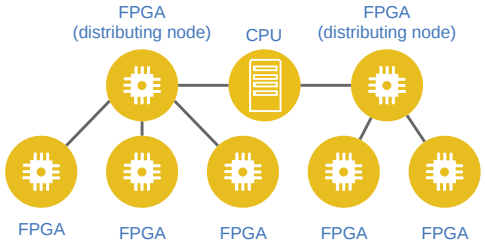

Fig. 5. Optimized topology for `MPI_Scatter/Gather` for 8 nodes.

average, despite the small sizes of the testcases and the unoptimized code. The power consumption of the larger ReH²PC clusters are one order of magnitude lower than the CPU only versions. The chosen example application requires significant synchronization between the nodes and consequently becomes communication bound at larger cluster sizes. The comparison of the different 1024x1024 test cases reveals that the ReH²PC turns earlier to communication bound then the CPU clusters. Increasing the nodes from 8 to 16 for the ReH²PC for the same workload decreases the performance, in contrast to the CPU only cluster. Hence, the FPGAs are also measurably faster in computation than CPUs. In the smaller 256x256 test case both clusters are communication bound.

We argued earlier for the use of MPI on ReH²PC clusters because of the low latency of the hardware communication stack in FPGAs. The comparison with the optimized tree topology highlights this important feature. The optimization of the structure (see Figure 5) is not reducing the bandwidth of the CPU node (as depicted in Figure 2), but the numbers of hand-shakes between the root CPU and the first layer of FPGA nodes. Additionally, some communication edges are parallelized. In this case, the comparison of the un-optimized vs. optimized 256x256/8 test cases reveals a speedup of 1.27 (2136.72 ms $\rightarrow$ 1686.55 ms) for the CPU and 1.32 (842.58 ms $\rightarrow$ 637.87 ms) for the ReH²PC cluster. All versions of the 256x256/8 test case are communication bound and therefore the parallelization of communication increases performance for the CPU only and ReH²PC cluster. However, the observation that the even more communication bound ReH²PC cluster — because the FPGAs compute faster — has a higher speedup due to the tree optimization can be attributed to the simple fact that FPGAs and not CPUs were used as distributing nodes (see Figure 5). Consequently, if the FPGAs were not faster in processing the synchronizing handshakes, the ReH²PC cluster would have the same speedup like the pure CPU cluster and would be approximately 46 ms slower in each iteration. Thus, the offloading of communication to the FPGA nodes is increasing the performance additionally.

```
#include "mpi.h"
int main() {
  int rank, size;
  MPI_Status status;
  MPI_Init();
  MPI_Comm_rank(MPI_COMM_WORLD, &rank );
  MPI_Comm_size(MPI_COMM_WORLD, &size );
  //... data initialization ...
  int local_grid[local_dim+2][DIM];
  for(int l = 0; l < max_iterations; l++) {
    MPI_Scatter(&grid[0][0], local_dim*DIM, MPI_INTEGER, &local_grid[start_line][0],
 local_dim*DIM, MPI_INTEGER, 0, MPI_COMM_WORLD);
    //exchange halo regions
    if(rank == 0) {
      for(int r = 1; r < size; r++) {
        int first_line = r*local_dim;
        int last_line = (r+1)*local_dim -1;
        if(r != 0)
          first_line--;
        if(r != size-1)
          last_line++;
        MPI_Send(&grid[first_line][0], DIM, MPI_INTEGER, r, 0, MPI_COMM_WORLD);
        MPI_Send(&grid[last_line][0], DIM, MPI_INTEGER, r, 0, MPI_COMM_WORLD);
      }
      for(int i = 0; i < DIM; i++)
         local_grid[border_endline][i] = grid[border_endline-1][i];
    } else {
      MPI_Recv(&local_grid[halo_startline][0], DIM, MPI_INTEGER, 0, 0,
MPI_COMM_WORLD, &status);
      MPI_Recv(&local_grid[halo_endline][0], DIM, MPI_INTEGER, 0, 0, MPI_COMM_WORLD,
 &status);
    }
    for(int i = 1; i < local_dim+1; i++) {
      for(int j = 0; j< DIM; j++) {
        if( (i == 0 && absoulte_start == 0) || (i == local_dim && absolute_end ==
DIM-1) || j == 0 || j == DIM -1)
          local_new[i][j] = local_grid[i][j];
         else
          local_new[i][j] = (local_grid[i][j-1] + local_grid[i][j+1] + local_grid[i
-1][j] + local_grid[i+1][j])/4.0;
      }
    }
    MPI_Gather(&local_new[result_start_line][0], local_dim*DIM, MPI_INTEGER, &grid
[0][0], local_dim*DIM, MPI_INTEGER, 0, MPI_COMM_WORLD);
  }
  //...result verification...
  MPI_Finalize();
  return 0;
}
```

Listing 5. Code snippet for the Jacobi 2D application.

Finally, the resource usage of the ZRLMPI run-time module within the FPGA is in the range of 1 – 5 % of the total available resources.

In summary, our proof of concept implementation is able to leverage the latency advantages of FPGAs with negligible resource overhead while providing a one-click solution for the user. Roughly speaking, using ZRLMPI on a large ReH^{2}PC cluster gets the very same job done in ¼ of the time (speedup 2.0 – 5.6) and consumes only ⅒ of the power (9% – 53%), compared to a pure CPU cluster.

VI. Related Work

In 2006, a group from the University of Toronto started to implement their own lightweight MPI version, called TMD-MPI, with a focus on the ReH^{2}PC domain [17], [29]. Their work starts with a new design flow to deploy an MPI application on multiple FPGAs. They introduce their PCIe-based communication network and a "Message Passing Engine" to map the MPI behavior to their environment. The authors implemented a software and a hardware version of their library, which supports some very basic MPI calls (i.e. `MPI_Send` and `MPI_Recv`). Finally, they demonstrated the functionality of their platform using the heat equation and compared the results to PowerPC and MicroBlaze nodes. The team in Toronto continued their research with their recent *Galapagos* cluster and provided network and PCIe abstractions and a method to virtualize the FPGA [13][14]. The cluster consists of six FPGAs, but they did not continue the support of MPI.

In a recent effort, *Naylor et al.* presented "Tinsel", an FPGA-centered hyper-threaded RISC-V framework and a custom FPGA cluster with 12 Stratix V [15]. Tinsel uses a processor overlay with a full custom network on chip interconnect to support event-driven parallel programming models. Their messaging routines look similar to the MPI functions, but rely on the custom 2D mesh communication arrangement and consequently make the generalization of Tinsel difficult.

Also in 2019, *De Matteis et al.* from the ETH Zurich presented a "Streaming Message Interface" as ReHPC programming model [12]. They target eight Stratix 10 FPGAs connected in a 2D torus, so that each FPGA can only reach four other FPGAs. Hence, message forwarding and routing for this custom topology is part of their framework. The authors argue that due to pipelining and vectorization, FPGAs are not compatible with the classical MPI API and thus, the "Streaming Message Interface does not assume that buffers are first computed and then communicated — instead, sending a message is integrated into the pipeline." [12, p. 2]. They design a new API for sending streams that is heavily inspired by MPI itself, but optimized for pipelining within the FPGA. For a message of size one, their API is identical to the original MPI. Consequently, their framework can only be used for pure FPGA clusters.

A completely different approach was pursued by *Gao et al.* in [26]. To accelerate HPC clusters, they offload only the `MPI_Barrier` routine to the bus-attached FPGA and can achieve a significant reduction in latency for the execution of the barrier. Very recently, a group from the Boston University followed a similar way [32] and related technologies can also be found as part of commercial products like [27]. However, this approach offloads part of the MPI run-time environment to FPGAs but it is not accelerating the actual computation workload.

VII. Conclusion

The rising demand for energy-efficient compute power in HPC environments leads to a wider adaption of reconfigurable hardware. With our proof of concept we wanted to exhibit the potential of MPI to become a standard for Reconfigurable Heterogeneous High-Performance Computing clusters. We show that this goal can be achieved if the FPGAs are directly attached to the interconnection network of the cluster and the application code is transpiled using the final cluster description. Our results show that the optimization for a specific cluster setup at compile-time can leverage FPGAs in HPC clusters. Furthermore, the heterogeneous nature of ReH^{2}PC offers new ways to optimize collective operations for additional performance increases. The presented proof of concept implementation of MPI for ReH^{2}PC requires further research to increase the coverage of MPI routines, optimize the handshake implementations and to also include design-space exploration of the FPGA parts. Finally, we hope that our approach to augment the MPI standard with a transpiler that auto-tunes applications for particular cluster setups will bring FPGAs and CPUs to work together efficiently from a single source of code.

References

[1] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49, Taipei, Taiwan: IEEE Press, 2016, 7:1–7:13.

[2] F. Abel *et al.*, "An FPGA platform for hyperscalers," *Proceedings - 2017 IEEE 25th Annual Symposium on High-Performance Interconnects, HOTI 2017*, pp. 29–32, 2017. DOI: 10.1109/HOTI.2017.13.

[3] B. Ringlein *et al.*, "System Architecture for Network-Attached FPGAs in the Cloud using Partial Reconfiguration," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Barcelona, Spain: IEEE, 2019, pp. 293–300. DOI: 10.1109/FPL.2019.00054.

[4] M. Saldaña and P. Chow, "TMD-MPI: An MPI implementation for multiple processors across multiple FPGAS," *Proceedings - 2006 International Conference on Field Programmable Logic and Applications, FPL*, pp. 329–334, 2006. DOI: 10.1109/FPL.2006.311233.

[5] V. Rana *et al.*, "Partial dynamic reconfiguration in a multi-FPGA clustered architecture based on linux," in *Proc. IEEE Int. Parallel and Distributed Processing Symp*, Mar. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370363.

[6] J. P. Walters *et al.*, "Mpi-hmmer-boost: Distributed fpga acceleration," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 48, no. 3, pp. 223–238, Sep. 2007. DOI: 10.1007/s11265-007-0062-9.

[7] M. Showerman *et al.*, "Qp: A heterogeneous multiaccelerator cluster," in *In Proc. 10th LCI International Conference on High-Performance Clustered Computing – LCI'09, 2009. Memory reliability*, 2009.

[8] M. Porrmann *et al.*, "Raptor–a scalable platform for rapid prototyping and fpga-based cluster computing," *Parallel Computing: From Multicores and GPU's to Petascale, Advances in Parallel Computing*, vol. 19, 2010.

[9] X. Y. Niu, K. H. Tsoi, and W. Luk, "Reconfiguring distributed applications in FPGA accelerated cluster with wireless networking," in *Proc. 21st Int. Conf. Field Programmable Logic and Applications*, Sep. 2011, pp. 545–550. DOI: 10.1109/FPL.2011.106.

[10] Y. Kono, K. Sano, and S. Yamamoto, "Scalability analysis of tightly-coupled FPGA-cluster for lattice boltzmann computation," in *Proc. 22nd Int. Conf. Field Programmable Logic and Applications (FPL)*, Aug. 2012, pp. 120–127. DOI: 10.1109/FPL.2012.6339275.

[11] A. Lawande, A. D. George, and H. Lam, "Novo-G#: a multidimensional torus-based reconfigurable cluster for molecular dynamics," *Concurrency Computation Practice and Experience*, vol. 28, pp. 2374–2393, 2015. DOI: 10.1002/cpe.3565.

[12] T. D. Matteis *et al.*, "Streaming Message Interface: High-Performance DistributedMemory Programming on Reconfigurable Hardware," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, Nov. 2019.

[13] N. Eskandari *et al.*, "A modular heterogeneous stack for deploying fpgas and cpus in the data center," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: ACM, 2019, pp. 262–271. DOI: 10.1145/3289602.3293909.

[14] N. Tarafdar and P. Chow, "Libgalapagos: A software environment for prototyping and creating heterogeneous fpga and cpu applications," in *Proceedings of 6th International Workshop on FPGAs for Software Programmers*, VDE VERLAG GMBH - Berlin - Offenbach, 2019.

[15] D. T. Matthew Naylor Simon W. Moore, "Tinsel: A manythread overlay for fpga clusters," in *Proceedings of the 29th International Conference on Field Programmable Logic and Applications*, IEEE Catalog Number: CFP19623-ART, IEEE, 2019. DOI: 10.1109/FPL.2019.00066.

[16] Paderborn Center for Parallel Computing, *Annual Report 2017/18/19 – Paderborn Center for Parallel Computing*. Paderborn Center for Parallel Computing, 2019.

[17] A. Patel *et al.*, "A scalable FPGA-based multiprocessor," *Proceedings - 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2006*, pp. 111–120, 2006. DOI: 10.1109/FCCM.2006.17.

[18] B. Ringlein *et al.*, "ZRLMPI: A Unified Programming Model for Reconfigurable Heterogeneous Computing Clusters," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Fayetteville, Arkansas: IEEE, May 2020, p. 220. DOI: 10.1109/FCCM48280.2020.00051.

[19] A. Lastovetsky and R. Reddy, "Heterompi: Towards a message-passing library for heterogeneous networks of computers," *Journal of Parallel and Distributed Computing*, vol. 66, pp. 197–220, 2006.

[20] D. Clarke *et al.*, "Fupermod: A framework for optimal data partitioning for parallel scientific applications on dedicated heterogeneous hpc platforms," in *Parallel Computing Technologies*, V. Malyshkin, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 182–196.

[21] The Open MPI Project. (2020). "General run-time tuning — affinity," The Open MPI Project, [Online]. Available: https://www.open-mpi.org/faq/?category=tuning#using-paffinity-v1.3.

[22] The MPICH project. (2020). "Using the Hydra Process Manager — Process-core Binding," The MPICH project, [Online]. Available: https://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manager#Process-core_Binding.

[23] P. S. Pacheco, *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[24] M. Snir *et al.*, *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd. (Revised). MIT Press, 1998.

[25] J. Weerasinghe *et al.*, "Disaggregated FPGAs: network performance comparison against bare-metal servers, virtual machines and linux containers," *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, no. i, pp. 9–17, 2017. DOI: 10.1109/CloudCom.2016.0018.

[26] S. Gao, A. G. Schmidt, and R. Sass, "Hardware implementation of MPI_Barrier on an FPGA cluster," in *Proc. Int. Conf. Field Programmable Logic and Applications*, Aug. 2009, pp. 12–17. DOI: 10.1109/FPL.2009.5272560.

[27] Mellanox Technologies. (2020). "Fabric Collective Accelerator (FCA) – Product Brief," [Online]. Available: https://www.mellanox.com/related-docs/prod_acceleration_software/FCA.pdf.

[28] P. Balaprakash *et al.*, "Autotuning in high-performance computing applications," *Proceedings of the IEEE*, vol. 106, no. 11, Jul. 2018. DOI: 10.1109/JPROC.2018.2841200.

[29] M. Saldana *et al.*, "MPI as a Programming Model for High-Performance Reconfigurable Computers," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 3, no. 4, pp. 1–29, 2010. DOI: 10.1145/1862648.1862652.

[30] T. Hoefler and T. Schneider, "Optimization principles for collective neighborhood communications," in *Proc. Int. Conf. High Performance Computing, Networking Storage and Analysis SC '12:*, Nov. 2012, pp. 1–10. DOI: 10.1109/SC.2012.86.

[31] S. Vadhiyar, G. Fagg, and J. Dongarra, "Automatically tuned collective communications," IEEE, 2000, pp. 3–3. DOI: 10.1109/SC.2000.10024.

[32] Q. Xiong *et al.*, "Accelerating MPI Collectives with FPGAs in theNetwork and Novel Communicator Support," in *Proceedings of the 28th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Fayetteville, Arkansas: IEEE, 2020, p. 215. DOI: 10.1109/FCCM48280.2020.00046.