Accepted for IEEE IPDPS 2022

OmpSs@cloudFPGA: An FPGA Task-Based Programming Model with Message Passing

Juan Miguel de Haro*†, Rubén Cano*,
Carlos Álvarez*†, Daniel Jiménez-González*†,
Xavier Martorell*†, Eduard Ayguadé*†, Jesús Labarta*†
*Barcelona Supercomputing Center
†Universitat Politècnica de Catalunya
{juan.deharoruiz, ruben.cano, carlos.alvarez, djimenez,
xavier.martorell, eduard.ayguade, jesus.labarta}@bsc.es

Francois Abel, Burkhard Ringlein, Beat Weiss
IBM Research Europe
{fab, ngl, wei}@zurich.ibm.com

***Abstract*—Nowadays, a new parallel paradigm for energy-efficient heterogeneous hardware infrastructures is required to achieve better performance at a reasonable cost on high-performance computing applications. Under this new paradigm, some application parts are offloaded to specialized accelerators that run faster or are more energy-efficient than CPUs. Field-Programmable Gate Arrays (FPGA) are one of those types of accelerators that are becoming widely available in data centers.**

This paper proposes OmpSs@cloudFPGA, which includes novel extensions to parallel task-based programming models that enable easy and efficient programming of heterogeneous clusters with FPGAs. The programmer only needs to annotate, with OpenMP-like pragmas, the tasks of the application that should be accelerated in the cluster of FPGAs. Next, the proposed programming model framework automatically extracts parts annotated with High-Level Synthesis (HLS) pragmas and synthesizes them into hardware accelerator cores for FPGAs. Additionally, our extensions include and support two novel features: 1) FPGA-to-FPGA direct communication using a Message Passing Interface (MPI) similar Application Programming Interface (API) with one-to-one and collective communications to alleviate host communication channel bottleneck, and 2) creating and spawning work from inside the FPGAs to their own accelerator cores based on an MPI rank-like identification. These features break the classical host-accelerator model, where the host (typically the CPU) generates all the work and distributes it to each accelerator.

We also present an evaluation of OmpSs@cloudFPGA for different parallel strategies of the N-Body application on the IBM cloudFPGA research platform. Results show that for cluster sizes up to 56 FPGAs, the performance scales linearly. To the best of our knowledge, this is the best performance obtained for N-body over FPGA platforms, reaching 344 Gpairs/s with 56 FPGAs. Finally, we compare the performance and power consumption of the proposed approach with the ones obtained by a classical execution on the MareNostrum 4 supercomputer, demonstrating that our FPGA approach reduces power consumption by an order of magnitude.

***Index Terms*—FPGA, MPI, OpenMP, programming models, network-attached FPGA, stand-alone FPGA, High-Level Synthesis, heterogeneous programming, High-performance computing**

I. Introduction

As applications require more performance, the underlying hardware has to evolve to satisfy their needs. This is why heterogeneous architectures are becoming more popular recently. In these systems, part of the work traditionally done by a CPU is offloaded to a specialized device. This device, e.g., a GPU, FPGA, or ASIC, is more efficient than the CPU in some way, like performance or power. The most common heterogeneous architectures for High-Performance Computing (HPC) use GPUs. Recently, more and more FPGAs are making their way into data centers (DC) [1] to help optimize the compute as well as the data movement of cloud workloads [2]. An FPGA is a reconfigurable device that can host any hardware design that fits within its available resources. Therefore, FPGAs can be tuned and optimized for a wide variety of workloads. The vast majority of FPGAs deployed in DCs operate with a traditional CPU-FPGA bus attachment such as the Peripheral Component Interconnect Express (PCIe). In recent data centers, we observe the emergence of a new interconnection pattern in which FPGAs are directly connected to the DC network fabric. Examples of such a pattern include the FPGAs placed in between the CPU and the network in a so-called "*bump-in-the-wire*" configuration [3] and the FPGAs operated as standalone network-attached accelerators [2]. This is a complete change of paradigm in CPU-to-FPGA and FPGA-to-FPGA inter-communications. It opens new perspectives for the use and the deployment of clusters of FPGAs in heterogeneous cloud DCs. However, the specificities of network-attached FPGAs require a new framework to profit from the potential provided by the clustering of FPGAs over high-speed and low-latency networks.

In this paper, we introduce a programming model framework called OmpSs@cloudFPGA that automatically takes care of all the low-level infrastructure of such clusters. With it, the user can exploit the parallelism of large FPGA clusters and scale out an application with minimal changes to the code. The main objective of OmpSs@cloudFPGA is to scale with decent performance on big clusters. This paper presents the following contributions:

- OmpSs@cloudFPGA, a parallel task-based programming model that allows easy and efficient programming of FPGA clusters.
- A method to distribute program task control over the

FPGA nodes on the cluster to improve system scalability.
- Direct MPI-like FPGA-to-FPGA communication to alleviate host communication channel bottlenecks.
- A data directory that dynamically performs data copies between a CPU host and other types of hardware devices attached to it.
- An evaluation of the presented model on a cluster of 56 FPGAs with an N-body application, delivering the highest reported performance of N-body implemented over FPGA to the best of our knowledge.
- A comparison of the performance and power consumption of the presented model with the one obtained by a CPU cluster.

II. Background

A. FPGA programming

Traditionally, the complete design implemented on an FPGA had to be programmed in a Hardware Description Language (HDL), e.g., Verilog or VHDL. The Register Transfer Level (RTL) engineer would have to code a hardware module and connect it directly to the FPGA pins or use Intellectual Property (IP) libraries that already implement the necessary protocols. With the rise of High-Level Synthesis and automatic tools, the gap between RTL and software engineers has been dramatically reduced. Nowadays, the FPGA user can program a custom IP with high-level languages and build and test the design through an automatic framework. For example, the Xilinx Vitis platform allows synthesizing accelerators using C/C++ on the FPGA, and communicating with the software application with a provided API.

B. OmpSs

OmpSs is a task-based programming model [4] developed at the Barcelona Supercomputing Center. In OmpSs, the parallelism of a C/C++ application is expressed mainly with tasks, similar to OpenMP tasks. These tasks are pieces of code that may require or produce data, in the form of single memory addresses or regions of memory. The runtime of OmpSs is called Nanos5 [5]. It dynamically checks for dependencies among tasks, based on the data requirement and production, and executes all possible tasks in parallel on Symmetric Multiprocessors (SMP). The programming model of OmpSs is flexible and was extended to execute tasks in devices such as GPUs and FPGAs. In the case of FPGAs, the extension is referred to as OmpSs@FPGA.

C. OmpSs@FPGA

The OmpSs@FPGA framework [6] is the extension of OmpSs that enables executing FPGA tasks on heterogeneous CPU+FPGA-based systems. It uses FPGA-specific vendor tools to automate the generation of the FPGA bitstream from the original user source code written in C/C++. The supported HLS tool at the time of writing is Vivado HLS. Therefore, the language subset in FPGA tasks is limited by Vivado HLS, which does not support all features usually available in a regular C/C++ program due to the nature of FPGAs. For example, dynamic memory allocation (e.g., `malloc`) and recursive calls are not supported. Moreover, OmpSs@FPGA introduces some limitations too. The bit width of task parameters is limited to 64-bit because the communication protocol with the FPGA uses a single 64-bit word per parameter. Also, at the time of writing, the framework does not provide a mechanism to support return types other than `void`.

The compilation process is shown in Fig. 1. It consists of a source-to-source compiler, Mercurium, the Accelerator Integrator Tool (AIT), and the native compiler (e.g., GCC). Mercurium reads the original code and separates the code to be executed by CPUs from the code targeted at FPGAs. However, it does not check language compatibility with the vendor HLS tool, hence any compilation error related to this matter is reported by the tool itself. AIT receives the FPGA code and generates the final bitstream. The native host compiler links the user application with the Nanos5 runtime, linked to the Xtasks library. The latter implements the low-level communication between the specific host and FPGA in the system.

OmpSs@FPGA introduces new clauses to the `#pragma target` to specify that the code associated with the task is to be executed on the FPGA, along with other device-specific clauses. For example, the user can declare local arrays, which are buffers stored in the FPGA local memory, e.g., SRAMs embedded in the FPGA fabric. Mercurium identifies FPGA tasks, extracts the necessary code, applies some transformations, and generates new source code that includes the code of the tasks. An HLS tool synthesizes this code into one or more hardware IP accelerators.

The user code is instantiated into a wrapper that communicates with a hardware runtime inside the FPGA bitstream, called Picos OmpSs Manager (POM). AIT automatically connects POM with the user-defined accelerators and with the CPU at compile time. At runtime, POM directly manages the accelerators and communicates with the CPU host of the device. Each time a CPU thread creates an FPGA task and is ready for execution, the Xtasks library sends the task information directly to POM through a hardware queue. Accelerators can also create CPU and FPGA tasks, and POM can handle the dependencies they may have. Therefore, POM is the counterpart of the Nanos5 runtime located in the host. The tasks spawned by POM remain local to the FPGA and are not visible by the runtime in the CPU. As a result, the number of communications between the host and the FPGA is reduced.

D. OmpSs-2

OmpSs-2 is an improved version of OmpSs. Although the programming model itself is very similar, OmpSs-2 provides a slightly different syntax, such as the use of pragma `oss` instead of `omp`. The significant differences relate to the Nanos6 runtime library [5], which comes with better performance, more and richer features than Nanos5. Therefore, we decided to work directly with OmpSs-2 and Nanos6 to develop our work.

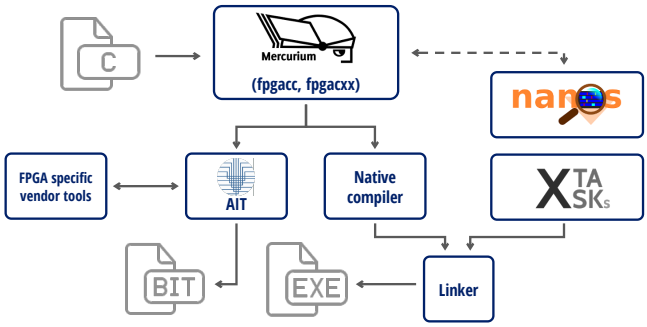

Fig. 1. Compilation process for OmpSs@FPGA

III. OmpSs@cloudFPGA

A. Modifications to the Nanos6 runtime

To perform the work presented we have extended the Nanos6 runtime with support for multiple FPGAs. Our main contribution in Nanos6 that is not present in Nanos5 is the data directory, used to keep coherency between devices connected to the same CPU host. Before a task is executed on a device, the directory registers the memory regions accessed by this task, extracted from the dependencies. Initially, Nanos6 only supported GPUs with CUDA unified memory. The GPU automatically performs copies when needed in this model, but only between CPU-GPU and GPU-GPU. Instead, the directory handles both GPU and FPGA devices even when connected to the same host, and performs device memory allocations, address translation, and data copies. If data accessed by a device task is present only in the host, the directory issues a memory copy from the host to the device. It also handles copies from device to host if a host task needs data modified by a device and copies between devices if a device task needs data from another device. When the host requests a `taskwait`, all device data regions are invalidated and thus copied back to the CPU. The directory transparently takes care of all the data copies without any explicit allocation and memory copies in the user code.

B. Extensions to OmpSs-2

With the directory approach, the host has to manage all tasks and data movements between all devices. Although this feature is valuable and easy for the programmer to use, as the number of FPGAs increases, the host management of data movements and tasks may become a bottleneck. This is further discussed in section VI.

OmpSs@cloudFPGA extends the OmpSs-2 programming model and framework with two main features to solve the problem mentioned above.

1) FPGA point-to-point communication: To scale out applications on FPGA clusters, we first need to distribute the work spawn on each FPGA. This is achieved by the POM hardware runtime, which allows an FPGA to create tasks independently, without the interaction of a CPU. Next, we need a way to distribute the data movement between FPGAs.

To solve this issue, we propose an interface similar to the well-known MPI for FPGAs and a corresponding API that can be called from the user code. We called this FPGA version of MPI OmpSs MPI for FPGAs (OMPIF). The implemented API provides basic calls to send and receive messages. Each FPGA gets assigned a rank ranging from zero to the size of the cluster minus one. The CPU does not have a rank associated with the initial implementation as we have not implemented a software OMPIF runtime. I.e., all ranks are FPGA nodes, and the cluster size counts only the number of FPGAs. In future implementations, we plan to provide a software runtime that can communicate with the FPGAs through OMPIF.

Along with the rank and the size of the cluster, an FPGA accelerator can use the API to communicate with other devices. Like in classical MPI, a message comprises user data, and extra information called the envelope. The latter contains the source and destination ranks of the message and a user-defined tag. This envelope is used to route the message to its corresponding destination and identify it so a receive call can match it. The equivalent of an MPI communicator is fixed to the total number of available FPGAs. However, a distributed task could target a subset of the cluster and use communicators to communicate in those subsets. The prototypes of the send and receive calls are:

- `void` **`OMPIF_Send`**`(const void* data, int count, OMPIF_Datatype datatype, int destination, uint8_t tag, OMPIF_COMM communicator)`
- `void` **`OMPIF_Recv`**`(const void* data, int count, OMPIF_Datatype datatype, int source, uint8_t tag, OMPIF_COMM communicator)`

All parameters are equivalent to their MPI counterparts except for a few modifications. `tag` is restricted to an 8-bit unsigned integer. Currently, the communicator is only allowed to have the `OMPIF_COMM_WORLD` value that involves all the FPGAs in the cluster. The API calls and specification are proposals and, therefore, subject to change. Both send and receive operations are blocking: `OMPIF_Send` returns when the send buffer is safe to be modified, and `OMPIF_Recv` returns when the buffer contains the matching message data.

2) Distributed task spawn: Up to this point, FPGA accelerators in a cluster can create tasks and communicate with each other through the OMPIF API. However, there is no way to easily start executing an application on the whole cluster from the host. The main objective is to provide the user with a simple way to execute a distributed application in a cluster of any size. The classical MPI approach is to execute the same binary in the cluster, and depending on the rank, each node decides what to do. However, there are some limitations for FPGAs. One of the main problems is that the cluster is heterogeneous, with at least two types of devices: a CPU and many FPGAs. Both have different characteristics, features, and roles in the application. Thus they require very different codes. For example, FPGA devices may not have a disk attached, so the CPU must perform all

disk operations required by the application. Another problem is memory allocation. User accelerators are not supported by an operating system or virtual memory in the FPGA. Instead of using an external allocator, it is more straightforward if the CPU does all the allocations and communicates the addresses to the accelerators. With the proposed model and use case, there is no immediate benefit in allocating memory on the FPGA. In general, the CPU provides more features, and the FPGA is mainly used to perform calculations only.

In the OmpSs@FPGA model, introduced in section II-C, the FPGA code is already isolated inside a task, which is specified as a C/C++ function. We extended the FPGA task declaration pragma with a `distributed` clause. Distributed tasks can only be called from the application code running in the CPU. When invoked, a single instance is replicated for each device in the cluster. This task instance becomes the entry point of the application execution. A distributed task is implemented as a regular accelerator, and it can use OMPIF and create tasks based on its rank and cluster size. These properties can be read at runtime by calls to the OMPIF API, equivalent to `MPI_comm_rank` and `MPI_comm_size`.

The final problem to consider is distributing the initial data in the FPGAs and retrieving it back. The CPU does not implement OMPIF, so we can not use send/receive calls. Instead, we propose different methods, including well-known communication patterns as pragma clauses, only valid for distributed tasks.

- Broadcast: The specified range is broadcasted to all FPGAs in the cluster before executing the task.
- Scatter: The specified buffer is divided into equal chunks of the specified size, and each chunk is sent to a different FPGA before executing the task.
- Gather: When the execution finishes, the specified buffer is divided into equal chunks of the specified size, and each chunk is read from a different FPGA.
- Send/receive: The specified range is sent to the specified rank before executing the task, or received after the execution finishes.

Furthermore, we provide a blocking Nanos6 API, `nanos6_distributed_memcpy` that enables the functionality of the clauses above at the application level.

C. Execution flow

To summarize the OmpSs@cloudFPGA model, Fig. 2 shows a high-level view of an OmpSs@cloudFPGA cluster and Listing 1 shows a code example. In Listing 1, vectors `a` and `b` are integer vectors divided in blocks of four elements. The total size of `a` and `b` is proportional to the number of nodes in the system since they are distributed with a scatter/gather collective. Each device computes some code over a chunk of `b` with a task per block. The result is sent to the next FPGA that receives this data, stores it in its part of `a` and computes another chunk of `b`. The CPU executes the `main` function (Main box in Fig. 2), initializes the data, and distributes it to the whole cluster with the scatter/gather clauses in the task pragma. Alternatively, the user could use

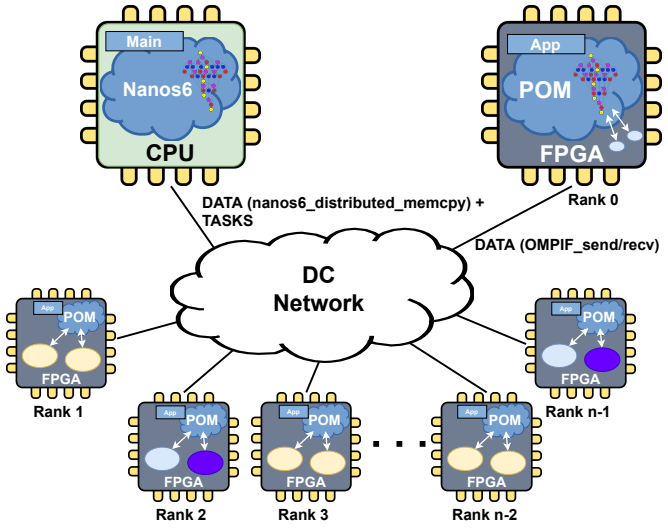

Fig. 2. High-level view of an OmpSs@cloudFPGA cluster

```
#pragma oss task device(fpga) in([4]a) out([4]b)
void fpga_comp_code(int* a, int* b) {...}
#pragma oss task device(fpga) distributed \
    scatter([nblocks*4]a) gather([nblocks*4]b)
void fpga_creator_code(int* a, int* b, int nblocks) {
    int rank = OMPIF_comm_Rank(OMPIF_COMM_WORLD);
    int size = OMPIF_comm_Size(OMPIF_COMM_WORLD);
    for (int i = 0; i < nblocks; ++i)
        fpga_comp_code(a + i*4, b + i*4);
    #pragma oss taskwait
    OMPIF_Send(b, nblocks*4, OMPIF_INT,
        (rank+1)%size, 0, OMPIF_COMM_WORLD);
    OMPIF_Recv(a, nblocks*4, OMPIF_INT,
        (size+rank-1)%size, 0, OMPIF_COMM_WORLD);
    for (int i = 0; i < nblocks; ++i);
        fpga_comp_code(a + i*4, b + i*4);
    #pragma oss taskwait
}
int main() {
    int *a, *b;
    int nblocks;
    ...//Initialize input with nblocks*4*nnodes elements
    fpga_creator_code(a, b, nblocks);
    #pragma oss taskwait
    ...//Process output
}
```

Listing 1: Example of OmpSs@cloudFPGA C code

the `nanos6_distributed_memcpy` API before calling the distributed task and after the `taskwait`. The CPU then creates a distributed task that Nanos6 broadcasts to all FPGAs. In the example, they execute the `fpga_creator_code` (App box in Fig. 2) function. This code communicates with its neighbor ranks and spawns `fpga_comp_code` tasks that POM handles. In Fig. 2, these tasks are represented by the colored circles in each FPGA node. When all FPGAs finish, the CPU program retrieves data from the cluster and processes the output.

IV. Implementation of OmpSs@cloudFPGA

This section describes the implementation details of our framework concerning a specific cluster of standalone

network-attached FPGAs. Such a cluster is accessible at the IBM Research laboratory in Zurich, Switzerland [7], [8], [9], and was made available for us to carry out this work. In the remainder of this article, we will refer to this cluster as cloudFPGA.

Despite the fact that the presented implementation targets a specific platform, most of its components and the programming model can target other clusters. The programming model is agnostic of the underlying platform where the tasks are executed.

A. The cloudFPGA system

cloudFPGA is a research cluster that aims to demonstrate new concepts and techniques for deploying FPGAs at a large scale in DCs. The system is built on three mains pillars: 1) the use of standalone network-attached FPGAs cards, 2) a hyperscale infrastructure for deploying such FPGA cards at large scale and in a cost-effective way [7] and, 3) an accelerator service that integrates and manages the standalone network-attached FPGAs in the Cloud [8].

The main difference with other research projects and commercial products is the absence of a PCIe bus for the host or another device to interact with the FPGA. Instead, this classical communication channel and its associated card driver are replaced with two UDP/IP and TCP/IP network stack interfaces and their affiliated socket programming models.

Similar to many other FPGA cloud offerings, cloudFPGA standalone network-attached FPGA builds on the common Shell-Role Architecture (SRA) design pattern [9]. This design separates the platform-specific parts (i.e., the Shell) from the application-specific parts (i.e., the Role) to increase the reusability and isolate the two parts. The Shell contains all necessary I/O components, and the network stack that hooks the FPGA to the DC network, as shown in the leftmost part of Fig. 3. It further abstracts all these hardware components by exposing standard AXI-stream interfaces to the user. From a computer operating system perspective, the Shell can be seen as the conceptual counterpart of the kernel space. The Role is the application-specific part of the FPGA logic. It embeds the user's custom application and can be assimilated to a CPU application executed in user space.

B. cloudFPGA Role architecture

With OmpSs@cloudFPGA, we add a new abstraction layer to the user in the Role. From the programmer's perspective, the application is built with hardware accelerators, usually in HLS C/C++. These accelerators are invoked from tasks created by other C/C++ code in the host or the FPGA and can communicate with other nodes in the cluster through the OMPIF API. In our framework, the cloudFPGA Role provides all the necessary components to make this possible. It contains mainly the OmpSs@FPGA hardware runtime (POM), the application accelerators, the message passing runtime (message sender/receiver), the packet decoder/encoder, and the memory manager. All these components are interconnected, as shown in the rightmost part of Fig. 3. External communication is done using UDP protocol because it has lower latency than TCP, and packet loss rate stays in acceptable bounds. There are many types of packets transported with UDP, used by the FPGA and the host. They can be classified according to the format of the header and the destination port:

- CPU commands: These commands are used by the host to control the Role, for example, to execute a task or read/write FPGA memory.
- CPU command responses: Most CPU commands expect one or more responses from the FPGA. The data contained in the packets depends on the command.
- Application messages: Packets generated by the use of the OMPIF API. There are two subtypes:
  - Data messages: Contain the actual data that one FPGA sends to another through a send operation.
  - *Ack* messages: Do not contain any relevant data, but are used to confirm to the sender that one or more messages have been successfully received. This is needed because UDP is not a reliable protocol and thus some packets can be dropped in the network.

1) Packet decoder and encoder: These modules are the entry and exit points of the Role for external communication. The decoder captures all incoming messages and forwards them to the corresponding module depending on the packet type. The decoder is also responsible for replying to data messages with an *ack*.

The packet encoder forwards messages sent by any module of the Role to the Shell. It also sets the destination node and port and adds a header for responses to CPU commands.

Both modules include debugging counters that track the number of packets sent/received, classified by the aforementioned types and subtypes. There are also registers that track the number of application messages exchanged with each individual node in the cluster. The host can access them with a CPU command.

2) OMPIF message passing runtime: As shown in Fig. 3, the runtime is a collection of two modules to handle message emission and reception, managed by POM. Thus internally, user accelerators issue send/receive petitions with the same interface used to create tasks. The message sender handles send operations only. It reads the memory in the specified address and transmits the data to the packet encoder. The message receiver reads a temporary buffer where the messages are stored just after arriving at the Role. When a message matching a receive request is found, it copies the data to the specified address.

3) Memory manager: This module handles memory read/write requests directly from the host and write requests to store data messages in a temporary buffer. Mainly it translates an AXI-stream codified with a specific format to multiple AXI-stream interfaces with another format. These interfaces communicate with a data mover IP, which receives read/write commands and access actual memory through an AXI4 interface.

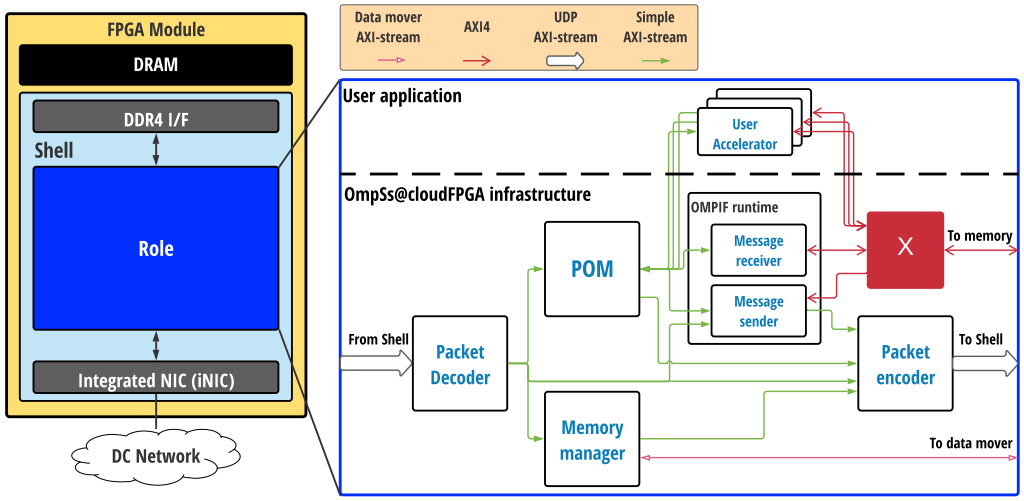

Fig. 3. cloudFPGA Shell-Role architecture, and detailed view of the OmpSs@cloudFPGA Role.

C. Message passing and reliability protocol

The designed protocol is used to control send/receive communication between two nodes. All FPGAs reserve a region of their memory address space to store temporarily data messages. When the user code issues a matching receive call, the data message is moved to its final address. In the proposed implementation, the reserved region is a circular queue stored in the board DRAM. To simplify the protocol, it is assumed that there is enough space in the reserved buffer to store all possible messages. This way, the sender can assume that all packets that arrive at the destination can be stored, and the receiver does not have to check if the buffer is full. We plan to remove this limitation in the future.

The underlying network that moves a message between two FPGAs has a specified Maximum Transmission Unit (MTU) of 1450 bytes, but the API supports sending more than this quantity. To overcome this limitation, the sender engine splits a message in frames of 1408 bytes. This size is the maximum multiple of 64 bytes that the payload can carry. The size is limited by a restriction in the memory address alignment. It has to be aligned to 64 bytes because the AXI4 data channels of the memory controller are 512-bit wide. Using unaligned addresses increases the resource usage and the complexity of the code. Therefore we decided to keep all addresses aligned.

The message passing protocol also takes into account packet loss. Because of UDP, data messages can be dropped before reaching their destination. The proposed solution is to use an acknowledge message to confirm that the data reached the destination. If this *ack* message is not received in a fixed amount of time, the sender retransmits the data message. To increase the bandwidth, a fixed window of four packets can be sent before waiting for an *ack*. If any of the four messages or the *ack* is dropped, the sender transmits the whole window again.

D. Host runtime implementation

1) Xtasks backend: The first step is to build a new backend for the Xtasks library, modifying the API specification. The old API assumes that only one device is attached, so there is no means to specify a target FPGA index. With this issue solved, we implemented a backend that uses UDP sockets to do all communication. They use two ports, one to send/receive tasks and another to read/write device memory. The library is thread safe and supports concurrent communication between different devices. Nevertheless, packets targeting the same device are serialized due to the protocol used to exchange information.

2) Nanos6 runtime: To support distributed tasks, we added a new type of virtual device in Nanos6 called broadcaster. This device is in fact the host, which receives a single distributed task and sends it to all devices in the cluster. This implementation is helpful because it uses an abstract class representing a device, and therefore can potentially work with any device, not only FPGAs. The broadcaster is also responsible for starting memory copies since the directory is unaware of the data locality once it belongs to the broadcaster.

V. Test application: N-body

The N-body application simulates the dynamic interaction of particles influenced by the force of gravity. Each particle has a position, velocity and mass associated, and the force between each pair of particles is calculated following the Newton's law of gravity:

$$F_{ij} = \frac{G \times m_i \times m_j \times (p_j - p_i)}{||p_j - p_i||^3}$$

Where F_{ij} is a 3-dimensional vector with the forces between particles i and j, m_i is the mass of particle i, p_i is a 3-dimensional vector with the position of particle i and G is the gravitational constant.

The simulation is an iterative process that calculates the forces between all pairs of particles on each iteration, accumulates the forces for each particle, and then updates the

position and velocities using the Euler method. Therefore, the algorithm has $O(n^2)$ time complexity.

A. Parallelization strategy

From the described application, we can extract two task types: one to calculate the interaction of particles and another to update the positions and velocities. The first step to parallelize the N-body is to distribute the particles in blocks of a fixed size. Then, each task operates on the block granularity. For example, to calculate the force interactions of four blocks, we need to execute the first task 16 times. To update the positions and velocities, we only need to execute the second four times. Each task has dependencies on the blocks that it reads or writes.

B. FPGA bitstream configuration

Our implementation of the N-body has been tested on Xilinx Kintex Ultrascale FPGAs of the cloudFPGA cluster. More details on the boards are discussed in section VI-A. Regarding the application, each task type can be implemented as an accelerator that can be instantiated multiple times in the FPGA fabric. Local memory, mainly Xilinx Block RAMs (BRAM), can be exploited because the block size is fixed. The accelerator first loads the block data in local memory then executes the algorithm using this memory, and finally stores the result back in the main memory. The force calculation accelerator is also optimized to take advantage of the FPGA properties and thus increase performance. The algorithm loop is unrolled by a factor of n and then pipelined with an initiation interval of one. That implies that n forces are calculated and accumulated on each cycle.

In the tested bitstream, we fit four instances of the force calculation accelerator and one instance to update the forces. The block size is 2048 particles, and the force calculation accelerator calculates 8 forces per cycle. The resulting throughput is 32 forces per cycle at 200MHz.

C. Task creation and communication implementation

1) First version, host-centric: The critical task of the N-body is force calculation. Because of its quadratic growth, most of the time is spent calculating pairs of forces. Therefore, our objective is to distribute this work between all FPGAs in the cluster. In our implementation, each device calculates only a subset of the forces. E.g., with 10 blocks of particles and 2 FPGAs, each one calculates the forces of 5 blocks against all the 10 blocks. That is a total of 50 tasks per device out of 100. However, both FPGAs need all the particle's positions to calculate the forces.

With the initial OmpSs-2 FPGA support, we can achieve this distribution of work with host task creation. Because the CPU handles all tasks, this one has a global view of the data locations. Thus the directory is able to maintain all accesses in the FPGAs coherent. To achieve coherency, the Nanos6 runtime has to move particle positions from host to FPGA and vice-versa. Nevertheless, this approach doubles the amount of data movements because to copy a block from one FPGA to another, the data must pass through the CPU memory first.

Algorithm 1 Pseudocode of the FPGA accelerator that creates tasks and communicates with the cluster

```
start ← (n/size) * rank
end ← start + n/size
for t in 0..timesteps do
    for i in 0..n do
        for j in start..end do
            calculate_forces(part_i, part_j, forces_j)
        end for
    end for
    taskwait()
    OMPIF_Allgather(forces_{0..n})
    for i in 0..n do
        update_particles(part_i, forces_i)
    end for
end for
```

Although this approach is easier to implement for the user, it has some limitations. The scalability depends heavily on the host throughput for both sending tasks and moving data between devices.

2) Second version, host-centric with distributed communication: This version takes advantage of the OMPIF runtime available in the bitstream. To move a block of data from one FPGA to another, Nanos6 calls a send operation on the device with the most recent data and a receive operation on the device that needs to update. With this method, the host throughput does not affect the application and the memory movements are distributed. However, the host still has to process all tasks of the application.

3) Final version, distributed communication and control: The main improvement of this implementation is that the FPGAs distributively handle both tasks and data movements. The bitstream includes an accelerator devoted to create tasks and call the OMPIF API, following algorithm 1. The function calls *calculate_forces* and *update_particles* are non-blocking task spawn functions, whereas *OMPIF_ALLGATHER* is a blocking call. In the pseudocode, *n* is the number of blocks, *rank* is a device index starting at 0, and *size* is the size of the cluster. Each device spawns only the tasks for its own accelerators and handles the dependencies using POM. Forces are sent after all force calculation tasks have finished. There is also no overlapping with the tasks to update particles. However, particle update tasks from one timestep are overlapped with the execution of the force calculation tasks from the next timestep.

Despite that in previous versions, only positions were exchanged between devices, we decided to move forces instead for this version. The N-body data set is stored in two buffers, one with forces and another with particle properties, including positions. Therefore, positions are not stored in consecutive memory locations between blocks. The host runtime is able to copy the necessary regions between FPGAs, but the OMPIF API requires one consecutive buffer to send/receive data.

We use an equivalent operation to an *MPI_Allgather* with the force buffer. After the device has sent and received all necessary blocks, it updates all the particles' positions and

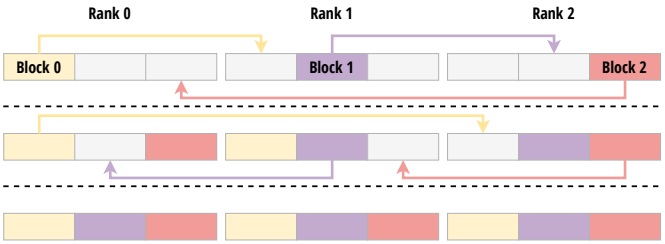

Fig. 4. Exchange of forces in N-body with the OMPIF allgather collective.

velocities. Although each FPGA does extra work, this part does not significantly affect performance due to the linear against quadratic time relation between the two task types.

The `allgather` collective is already supported by OMPIF, but only when the send buffer is a part of the receive buffer (equivalent in MPI to use the parameter `MPI_IN_PLACE`). Internally, each rank sends its part of the data to the next rank and increases the destination rank, going to 0 when overflowing until all ranks have been served. This process is illustrated in Fig. 4, in which there are three ranks and three blocks. Each rank has a reserved buffer to store all force blocks but calculates a single force block. Then, each node sends its own block to the other two, following the order of Fig. 4 to fill the missing blocks.

VI. Evaluation

We demonstrate the scalability of the OmpSs@cloudFPGA framework with the N-body application described in section V on a cluster of 56 FPGAs. We also perform a similar study with CPU nodes in the MareNostrum 4 supercomputer.

A. Experimental setup

The FPGA cards of the cloudFPGA platform that we used implement a Kintex Ultrascale FPGA (xcku060-ffva-1156-2-i) and 16GB of DDR4 memory. The Kintex is a mid-range FPGA chosen for its excellent performance-per-dollar ratio, a metric that is particularly relevant for a large scale deployment in the Cloud. The CPU host is an Intel Xeon E5-2640 v4 @ 2.40GHz processor with 24GB of RAM. The MareNostrum 4 supercomputer uses high-performance Intel Xeon Platinum 8160 CPUs @ 2.1GHz. Each node has 48 cores and 96GB of memory. For the experiments, we use the same task creation and communication as algorithm 1 but using OmpSs-2 instead of OmpSs@cloudFPGA and Intel MPI runtime (`MPI_allgather`) instead of OMPIF. The force calculation task is vectorized using Intel intrinsics of the AVX-512 extension. We built two versions of the application, the first using all node resources, with 512-bit vectors (16 forces) and 48 cores per node. The block size is 512 since it proved to achieve the best performance. Each CPU computes 768 parallel forces, which is 24 times more than the Kintex FPGAs. Therefore, we decided also to evaluate a more limited version using four cores with 256-bit vectors (8 forces), which is equivalent in vector length and number of computing units to the FPGA bitstream. We also used the same block size.

TABLE I
Kintex Ultrascale resource count, relative usage and max frequency of the cloudFPGA Shell and Role for the N-body

	LUT	FF	DSP	BRAM	LUTRAM	Fmax (MHz)
Resources	331K	663K	2760	1080	146K	
Usage (%)						
Bitstream	75.4	47.7	57.9	51.5	13.8	
Shell	34.4	19	0.65	27.8	7.5	
Accs	39.3	25.3	57.1	22.2	6.3	266
POM	0.62	0.35	0	1.1	0.08	219
OMPIF	0.2	0.2	0.18	0	0.01	304
Enc/dec	0.16	0.06	0	0.19	0.01	-

B. FPGA runtime and accelerator usage

Table I shows the number of resources for each primitive of the Kintex FPGA and also the relative usage of the tested bitstream. This resource usage is split between the Shell and the Role. For the latter, table I shows resource usage of the OmpSs@cloudFPGA components and the user accelerators. These are the POM and OMPIF hardware runtimes, packet encoder, and decoder. The table also reports the max frequency calculated after hardware synthesis. It is an estimation because the routing delay is not accurate. It depends on the place&route, which at the same time depends on the whole design. However, we can conclude that POM contains the critical path. Therefore we set the accelerator clock to 200MHz, giving a margin to the place&route algorithm for extra route overhead. Increasing the frequency requires a more detailed study of the internal design of POM and a change to its implementation with a more restrictive frequency target. The packet encoder and decoder have a fixed frequency of 156.25MHz because it is connected directly to a Shell clock.

C. Results

In all our tests, we fix the number of iterations to 16 and only change the number of particles to limit the number of possible input parameters. We also measure performance in Gpairs/s, which represents the number of force interactions calculated per second. The total number of particle pairs is $n^2 \times t$, where n is the number of particles and t is the number of iterations.

In our first experiment, we compared the performance of the three presented N-body versions in section V. With 16 FPGAs in the cluster, we ran the applications with the same bitstream, increasing the number of particles. Results are shown in Fig. 5. Host Tasks & Copies is the host-centric implementation in which the directory copies data from FPGA to host and again to another FPGA (section V-C1). Host Tasks+FPGA copies is the other host-centric implementation that uses the OMPIF runtime to copy data directly between FPGAs (section V-C2). FPGA Tasks+OMPIF refers to using the distributed task with FPGA task creation and the use of OMPIF from user code (section V-C3).

We can see that both host-centric implementations do not scale at the same rate as the distributed one. Also, the performance starts to stabilize at a much lower point. The

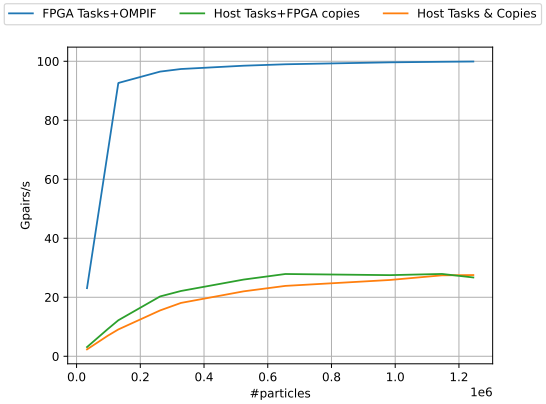

Fig. 5. Performance of the three FPGA versions of the N-body in a cluster of 16 FPGAs.

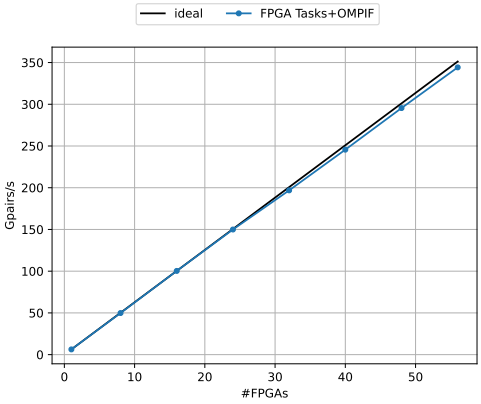

Fig. 6. Performance of the N-body application distributed over a cluster of FPGAs (61440 particles per FPGA)

distributed version is 4x faster. The host can only reach 26% of the peak performance with both copy strategies. This bottleneck worsens with the number of FPGAs used, leading to even worst comparable results with bigger clusters. Most of the performance loss is due to the communication latency between the host and FPGAs, there is a higher penalty since the data packets have to pass through more switches to reach the FPGAs. Even with only 16, the host cannot give work to each of the accelerators inside each FPGA, limiting its performance. For the data transfer, the protocol used by the host is similar to the one used by OMPIF. A fixed number of transfers is sent before waiting for finishing ack messages and sending the subsequent transfers. However, sending tasks from the host to the accelerators is more complex because the runtime does not know how many tasks it will send. Therefore, the runtime sends only one outstanding task per FPGA before waiting for the ack. Although it could be improved, this solution will not have significant effect in all applications because if tasks depend on each other, the runtime can not send outstanding tasks without breaking the dependence model. As we increase the number of FPGAs or the problem size, the runtime overhead becomes more critical for the host-centric versions. As it can be observed at the rightmost part of Fig. 5, the runtime overhead (suffered by both host-centric implementations but more acutely by the Host Tasks+FPGA copies version) becomes so large that it overcomes the data transfer overhead of the host-centric copies version.

Next, we investigated the scalability of the FPGA tasks+OMPIF version when increasing the number of FPGAs. Fig. 6 illustrates the performance of the application when varying the number of FPGAs from 1 to 56. The ideal plot represents the performance of a single FPGA multiplied by the cluster size. Our distributed implementation efficiency (actual performance compared to the theoretical maximum) is 98% for 56 FPGA nodes. The number of particles on each run also increases to give enough work to each FPGA. There are 30 blocks of 2048 particles per node (61440 particles). Although the number of particles is constant per device, the amount of work is not. As there are more FPGAs in the cluster, the number of tasks per FPGA also increases as well as the number of messages to send/receive. As seen in Fig. 6, with 56 FPGAs, we reach 344.22 Gpairs/s.

If we look at the single FPGA performance, we get 6.27 Gpairs/s. This is equivalent to a 97% efficiency over the theoretical peak performance of 6.4 Gpairs/s (32 forces per cycle at 200MHz). This 3% difference is caused by the memory movements of the forces between local and main memory, the update part of the particles, and the runtime overhead. We measured a more precise estimation of the overhead added by OmpSs@cloudFPGA with a different experiment by running the application with zero computation time. The results indicate that, on average, 60 cycles per task are spent on the hardware runtime (mainly dependence management), equivalent to 0.01% of a single force calculation task execution time. Hence, we can conclude that the runtime overhead is negligible.

We also measured the overhead introduced by OMPIF in a similar way. We ran the application without spawning computation tasks, only executing `OMPIF_Allgather`. The results show that we achieve a bandwidth of 4.2GB/s across 16 FPGAs out of 20GB/s (10Gbit/s ethernet per FPGA). The overhead added by data transfers is 0.32% of the total execution time. With the host-centric implementations, we measured an overhead of 54% with host copies and 50% with FPGA copies, caused by the bottleneck introduced when centralizing all transfers in a single device. Using FPGA copies reduces this overhead because the FPGAs perform the actual transfers, but the host still needs to communicate to start the copies. The host-FPGA latency is ≈ 0.5ms, sending a single message and receiving an ack is equivalent to 38% of the force calculation task time. Contrary to the FPGA runtime overhead, this time is not negligible and is one of the leading causes of the limited performance shown in Fig. 5.

Finally, we compare the best performance we have (FPGA tasks+OMPIF version and 56 nodes) with the equivalent execution in the MareNostrum 4 supercomputer. The results are shown in Table II. The MareNostrum vec256 row refers to

TABLE II
PERFORMANCE AND POWER COMPARISON OF THE N-BODY APPLICATION ON A CLOUDFPGA CLUSTER AND ON THE MARENOSTRUM 4 SUPERCOMPUTER, WITH 56 NODES AND 3440640 PARTICLES.

	Performance (Gpairs/s)	Efficiency	Watts	Perf./W
OmpSs@cloudFPGA	344.22	98%	840	0.41
MareNostrum vec256	211.81	94%	8673	0.024
MareNostrum vec512	2409.49	90%	15504	0.16

the limited resources implementation, it uses 256-bit vectors, 4 cores per node, and 2048 block size. The MareNostrum vec512 row is the version with full resources and uses 512-bits vector lengths, 48 cores per node, and 512 block size. With similar resources, the FPGA cluster proves to have 1.6 better performance than the CPU. However, the vec512 version is 7 times faster than the FPGA. Although in terms of efficiency, the FPGA version is above the other two. We also have to take into account that we are comparing a mid-range FPGA with a high-performance CPU. Moreover, the Kintex Ultrascale technology is four years older than the Intel Xeon Platinum CPUs. The Kintex uses 20nm TSMC lithography, while the Xeon uses Intel 14nm. Even with this disadvantage, the FPGA consumes one order of magnitude less power than the vec256 version and 18 times less than vec512. Looking at the performance per watt, the FPGA version is above both MareNostrum implementations.

The power consumption numbers for the FPGAs, 15 watts per FPGA, were measured in real-time by the cluster hardware. We can report that these numbers validate the power estimation generated by the Vivado tool for the considered bitstream. The MareNostrum power is taken from the Slurm manager reports, with 276 watts per node.

VII. RELATED WORK

There have been other efforts to implement the N-body on FPGAs and even ASICS. In [10], Sano et al. implement a full custom FPGA design that solves the N-body problem on a single Intel Arria10 FPGA, achieving 10.944 Gpairs/s. In [6], de Haro et al. uses OmpSs@FPGA to execute the N-body on a Xilinx Alveo U200 board reaching 37.62 Gpairs/s with a performance per watt of 0.58. Del Sozzo et al. [11] also presented a custom N-body implementation on a Xilinx Virtex Ultrascale+ board (VU9P). However, they only accelerate the force calculation and update the positions and velocities with software. They reach 13.441 Gpairs/s with a performance per watt of 0.672. ASICs have also been developed to accelerate the computation of the N-body. GRAvity PipE (GRAPE) [12] is an ASIC that computes the force interaction between particles, with 48 pipelines at 250MHz. The update part is still done in the CPU, which communicates with the chip through PCIe and an FPGA that works as a bridge. Its theoretical peak performance is 480 Gflops which translates to 24Gpairs/s, with a performance per watt of 0.5. Even though single-node performance of the Kintex is the lowest at 6.2 Gpairs/s our implementation with 56 nodes outperforms all other implementations. Applying our OmpSs@cloudFPGA framework to bigger or more modern FPGAs will result in even better results.

OpenMP also includes a mechanism to offload program regions to other devices with the `target` pragma. This feature has been used to accelerate OpenMP code with FPGAs. Knaus et al. [13] use the Clang compiler to extract the LLVM Intermediate Representation (IR) code of the `target` regions. This code is transformed to an RTL IP with the OpenCL HLS toolflow that FPGA vendors (e.g. Xilinx and Intel) provide. They feed the vendor tools directly with a custom IR instead of the original C/C++ code. The host code is modified to include calls to the OpenCL API. Sommer el al. [14] also modify the Clang compiler to extract the code in the OpenMP `target` regions to compile them with an HLS toolflow. However, instead OpenCL they use Xilinx Vivado HLS to synthesize the FPGA code. Then they use a custom toolchain, called ThreadPoolComposer (TPC), which similarly to AIT, generates automatically the whole FPGA design. To communicate with the host, they implement a `libomptarget` plugin that is used by OpenMP for the `target` regions. This library is linked to the TPC API that implements the low-level communication with the device. More related work for OpenMP FPGA offloading can be found in [15].

IBM has worked on an MPI implementation with FPGAs using the cloudFPGA platform [16]. Ringlein et al. implemented an MPI runtime, called ZRLMPI (with part of the MPI API), in both the FPGA and CPU. The ZRLMPI execution model is similar to MPI: all nodes have the same entry point to start execution. Although the same code targets both CPU and FPGA, it is optimized at compile-time depending on the executing device. Their proposal is not limited to only one CPU in the cluster, which can also participate in the application. Nevertheless, there is no task-level parallelism inside the FPGA. It can be seen as a single accelerator that executes HLS code. In the ZRLMPI approach, the cluster configuration must be known at compile-time, so the bitstream must be recompiled every time the configuration changes. With OmpSs@FPGA, the same bitstream and binary executable can be used for any number of FPGAs.

De Matteis et al. [17] also present an MPI-like model for FPGAs, more focused on streaming data from point-to-point rather than sending bulk messages. Nevertheless, they target a set of FPGAs connected with a specific topology, and message routing is part of their framework. They do not parallelize the application inside one FPGA using multiple accelerators.

Naylor et al. [18] introduce a RISCV-based multithreaded CPU for FPGAs and an FPGA cluster of 12 Stratix V. They present a message passing API similar to MPI to communicate data between cores of different FPGAs. Like De Matteis et al., they rely on a custom and fixed topology and hardware. They do not use hardware acceleration to execute the application, as it runs on the CPUs, but they accelerate global synchronization of nodes with hardware-assisted primitives.

There have been other extensions to the OmpSs programming model to support cluster offloading with tasks [19]. Sainz

et al. present a way to create MPI clusters dynamically and offload created tasks to a specific rank in a specific cluster. To achieve this, they use `MPI_Comm_spawn` to create and isolate each cluster and provide channels to communicate between clusters. However, they test their system on Intel Xeon Phi clusters, with up to 128 nodes.

Xiong et al. [20] introduce an FPGA implementation of the `MPI_Irecv` operation. They offload only the message matching part of the MPI runtime to the FPGA to accelerate MPI applications on CPUs. The architecture they propose does a similar job to the message receiver of OmpSs@cloudFPGA. Their implementation supports many outstanding receive requests using a two-level queue. However, we are more limited in resources because a big part of the FPGA is used for the user application, limiting the message receiver's capabilities.

VIII. Conclusion

With the introduction of FPGAs in data centers, a new paradigm has appeared for distributed parallel applications. In this work, we present extensions to an existing task-based programming model and a framework that can take advantage of the internal parallelism of an FPGA, as well as its inter and intra-communication capabilities to scale out parallel applications. With a single C/C++ code annotated with pragmas and a single bitstream compilation, the programmer can run an application in an FPGA cluster of any size. We have introduced an implementation of OmpSs@cloudFPGA, with support to an MPI-like API. We evaluated our framework on the IBM cloudFPGA research project equipped with mid-range network-attached FPGAs. The results show that the N-body application scales linearly to 56 nodes when implemented with OmpSs@cloudFPGA with near-perfect scalability. Furthermore, we have compared its performance and power consumption with the MareNostrum 4 supercomputer. The results show that even when using all the CPU resources, FPGAs still outperform CPUs' performance per watt. Furthermore, when comparing similar CPU and FPGA resources, the FPGA also provides better absolute performance than the CPU. In addition, the close to perfect linear scaling of FPGA accelerators shows the general advantage of the distributed task-programming approach to accelerate future high-performance applications at a low energy cost and with the same programmability as CPUs.

Acknowledgments

This work has been done in the context of the IBM/BSC Deep Learning Center initiative. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 754337 (EuroEXA), from Spanish Government (PID2019-107255GB-C21/AEI/10.13039/501100011033), and from Generalitat de Catalunya (2017-SGR-1414 and 2017-SGR-1328).

References

[1] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 13–24.

[2] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling fpgas in hyperscale data centers," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), Beijing, China, August 10-14, 2015*. IEEE Computer Society, 2015, pp. 1078–1086.

[3] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[4] A. Fernández, V. Beltran, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Task-based programming with ompss and its application," in *Euro-Par 2014: Parallel Processing Workshops*, 2014, pp. 601–612.

[5] (2022, February) BSC Programming Models, OmpSs and OmpSs-2 documentation and code. [Online]. Available: https://pm.bsc.es

[6] J. M. de Haro *et al.*, "Ompss@fpga framework for high performance fpga computing," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2029–2042, 2021.

[7] F. Abel, J. Weerasinghe, C. Hagleitner, B. Weiss, and S. Paredes, "An fpga platform for hyperscalers," in *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*, 2017, pp. 29–32.

[8] B. Ringlein, F. Abel, A. Ditter, B. Weiss, C. Hagleitner, and D. Fey, "System architecture for network-attached fpgas in the cloud using partial reconfiguration," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 293–300.

[9] IBM (2021) "cloudFPGA". [Online]. Available: https://github.com/cloudfpga

[10] K. Sano, S. Abiko, and T. Ueno, "Fpga-based stream computing for high-performance n-body simulation using floating-point dsp blocks," in *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, ser. HEART2017. Association for Computing Machinery, 2017.

[11] E. Del Sozzo, M. Rabozzi, L. Di Tucci, D. Sciuto, and M. D. Santambrogio, "A scalable fpga design for cloud n-body simulation," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–8.

[12] J. Makino and H. Daisaka, "Grape-8 – an accelerator for gravitational n-body simulation with 20.5gflops/w performance," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.

[13] M. Knaust, F. Mayer, and T. Steinke, "Openmp to fpga offloading prototype using opencl sdk," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 387–390.

[14] L. Sommer, J. Korinth, and A. Koch, "Openmp device offloading to fpga accelerators," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 201–205.

[15] F. Mayer, M. Knaust, and M. Philippsen, "Openmp on fpgas—a survey," in *OpenMP: Conquering the Full Hardware Spectrum*. Cham: Springer International Publishing, 2019, pp. 94–108.

[16] B. Ringlein, F. Abel, A. Ditter, B. Weiss, C. Hagleitner, and D. Fey, "Programming reconfigurable heterogeneous computing clusters using mpi with transpilation," in *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2020, pp. 1–9.

[17] T. De Matteis, J. de Fine Licht, J. Beránek, and T. Hoefler, "Streaming message interface: High-performance distributed memory programming on reconfigurable hardware," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. Association for Computing Machinery, 2019.

[18] M. Naylor, S. W. Moore, and D. Thomas, "Tinsel: A manythread overlay for fpga clusters," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 375–383.

[19] F. Sainz, J. Bellón, V. Beltran, and J. Labarta, "Collective offload for heterogeneous clusters," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, 2015, pp. 376–385.

[20] Q. Xiong, A. Skjellum, and M. C. Herbordt, "Accelerating mpi message matching through fpga offload," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 191–1914.