HORIZON 2020
TOPIC FETHPC-02-2017
Transition to Exascale Computing

Exascale Programming Models for Heterogeneous Systems
801039

D 2.2
Initial design document of MPI and GPI extensions for heterogeneous systems with distributed GPUs and FPGAs

WP 2: Programming high-performance communication networks and fabrics in heterogeneous systems

Date of preparation (latest version): 30/06/2019

DOCUMENT INFORMATION

Deliverable Number	D 2.2
Deliverable Name	Initial design document of MPI and GPI extensions for heterogeneous systems with distributed GPUs and FPGAs
Due Date	30/06/2019 (PM 9)
Deliverable lead	Fraunhofer ITWM
Authors	Valeria Bartsch (Fraunhofer ITWM) Daniel Holmes (EPCC) Timo Schneider (ETHZ)
Responsible Author	Valeria Bartsch (Fraunhofer ITWM) e-mail: `valeria.bartsch@itwm.fraunhofer.de`
Keywords	MPI GPI FPGA GPU
WP/Task	WP 2 /Task 2.2
Nature	R
Dissemination Level	PU
Planned Date	27/06/2019
Final Version Date	30/06/2019
Reviewed by	Olivier Marsden (ECMWF), Adrian Tate (CRAY)
MGT Board Approval	YES

DOCUMENT HISTORY

Partner	**Date**	**Comment**	**Version**
Fraunhofer	14/03	first version of GPI part for internal discussion	0.1
Fraunhofer	08/04	internally discussed structure added	0.2
Fraunhofer	25/04	new text for GPI added	0.3
ETHZ	17/05	new text for MPI-RMA for GPUs added	0.4
EPCC	24/05	test for persistent MPI collectives added	0.5
Fraunhofer	29/05/19	executive summary, introduction and summary	0.6
EPCC	30/05/19	added Terms and Definitions section 3.5	0.7
ETHZ	31/05/19	added Terms and Definitions section 2.6	0.8
ETHZ, FHG, EPCC	27/06/2019	changes after review	0.9
Fraunhofer	30/06/2019	Final cleanup, final version	1.0

Executive Summary

The document describes the initial design of MPI and GPI extensions for heterogeneous systems with distributed GPUs and FPGAs. It presents the design of MPI-RMA operations for distributed GPUs, MPI persistent collectives and GPI for distributed FPGAs.

We are building our support for heterogeneous HPC systems on the two widely used communication models MPI and GASPI (with its implementation GPI). Concerning heterogeneity we are tackling GPUs and FPGAs as compute elements and prepare MPI to offload compute for collectives to network hardware:

- For the support of MPI-RMA for GPUs we are targeting NVidia Volta GPUs. In the scope of the project a runtime library which supports a subset of the MPI specification and a launcher program to start MPI jobs across different GPUs has been designed and will be implemented.

- Persistent collectives in MPI aim to plan future usage of system resources by collective operations. The communication performance of key MPI persistent collective operations will be improved and overlap of communication and computation enabled. With the help of persistent collectives an offload of MPI collectives to NIC cards will be eased.

- Right now there is no notable MPI nor GASPI support for FPGAs. The proposed GPI support for distributed FPGAs will be light-weight and aided by the CPU. The target of this extension is FPGAs tightly connected to CPUs, possibly even with a shared memory.

In the scope of the EPiGRAM-HS project the presented extensions to MPI and GPI will be implemented and tested with at least one of the EPiGRAM-HS applications each.

Contents

1 Introduction

The document describes the initial design of MPI and GPI extensions for heterogeneous systems with distributed GPUs and FPGAs. EPiGRAM-HS aims to lower the threshold for the use of novel compute elements by integrating them more tightly into the HPC network stack. We have focused on GPUs and FPGAs because:

- GPUs have become available as accelerators in large HPC systems and are being used by many applications, e.g. in deep learning applications. Novel interconnect options are available, such as NVLink which allows peer-to-peer communication and the dCUDA technology which allows an individual identity (rank) for the GPU on the interconnect.

- FPGAs can be used either as accelerators for computing but also as active components in the network. They are showing a good latency and can be integrated so that a high bandwidth between the FPGAs is available. FGPAs have a good energy efficiency for many applications. The main hurdle for the use of FPGAs in HPC architectures is programmability. However, high Level Synthesis (HLS) tools are starting to become mature, making FPGAs easier to program.

Currently the integration of GPUs in the network stack is more sophisticated compared to FPGAs or any other accelerators (compare deliverable D4.1).

In the scope of EPiGRAM-HS, we are building our support for heterogeneous HPC systems on the two widely used communication models MPI and GASPI [1] (with its implementation GPI [2]). MPI and GASPI represent two different models for inter-node communication, namely message passing and partitioned global address space (PGAS). Traditionally, in message passing, the sender and the receiver are actively involved in the communication, while the PGAS concept exploits the RDMA (Remote Direct Memory Access) capabilities of HPC hardware with one-sided communication. MPI version 3.0 has incorporated some of the RDMA concepts blurring the boundaries between message passing and PGAS.

The work is split between integrating compute accelerators (GPUs and FPGAs) tighter in the network stack and preparing for FPGAs that take an active role in the network stack (e.g. as NIC cards on which MPI collectives can be offloaded):

- For the support of MPI-RMA for GPUs, we are targeting NVidia Volta GPUs. In the scope of the project, a runtime library that supports a subset of the MPI specification and a launcher program to start MPI jobs across different GPUs has been designed and will be implemented (see Section 2). MPI ranks are automatically distributed over GPU devices, i.e. a very tight integration of GPUs is the goal of this design. The MPI implementation will be tested by the Nek5000 application, which aims to run on GPUs. However, due to the large code of Nek5000, any rewrite is difficult and the proposed extension to the MPI implementation allows a transparent switch to GPU usage.

- For the offload of communication operations on compute-in-the-network devices, MPI collectives are being extended. Persistent collectives in MPI, described in

Section 3, aim to plan future usage of system resources by collective operations. The communication performance of key MPI persistent collective operations will be improved and overlap of communication and computation enabled. With the help of persistent collectives, offload of MPI collectives to NIC cards will be eased. The implementation will be tested with the help of the IFS, Nek5000, and iPIC3D applications within EPiGRAM-HS, which use collectives repeatedly in critical parts of their application and can benefit from speedups provided by an efficient planning of the collectives. In the case of Nek5000, the MPI-RMA extension for GPUs and implementation of persistent MPI collectives can be used together combining the potential speedups of both MPI extensions.

- Right now there is no notable MPI nor GASPI support for FPGAs. The proposed GPI support for distributed FPGAs will be light-weight and aided by the CPU. The target of this extension is FPGAs tightly connected to CPUs, possibly even with a shared memory (see Section 4). We are intending a loose level of integration of FPGA devices in distributed communication. Our goal is to support zero-copy data transfers for some scenarios. The main use-case is for the deep learning applications of EPiGRAM-HS. Due to their energy efficiency, FPGAs are an interesting hardware platform for deep learning applications.

2 MPI for distributed GPUs

2.1 Objective

GPUs have received a large amount of attention in HPC in recent years. Several Top500[1] machines offer GPUs, and the high peformance and energy efficiency of these devices is leveraged by many HPC applications. In Figure 1 we compare the power efficiency and peak performance of modern CPUs and GPUs. Peak performance of a V100 is 20x higher than that of a Core i7 8700K CPU, while its performance per Watt is higher by a factor of 3.6. We also show numbers for a low-power laptop CPU. Thus porting applications to GPUs is imperative.

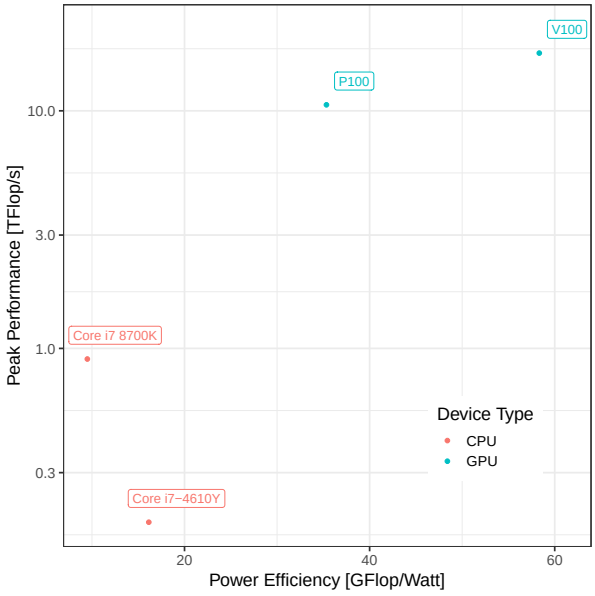

Figure 1: A comparison between GPU and CPU power efficiency and peak performance.

However, porting of traditional applications to GPUs is challenging. One approach to ease the burden of porting for the user are compilers which automatically offload parts of the computation to GPUs, i.e., PPCG. The main challenge here is the massive amount of parallelism offered by GPUs: GPUs are able to execute thousands of threads in parallel, given that these threads execute similar instruction sequences. HPC applications are often written for multicore CPUs, which have 1-2 orders of magnitude less parallel execution units but do not pose restrictions on the executed instruction sequences. Thus, auto-parallelizing compilers only work for a subset of HPC codes, i.e., kernels representable in the polyhedral model, and often provide less performance than manually ported code. However, manual porting is a lot of effort, i.e., a port of the WRF climate code took 3 months to complete [3], thus frameworks which make it easier to harness the compute power and energy efficiency offered by GPUs are useful for the scientific community.

The Message Passing Interface (MPI) is the de-facto standard to run HPC applications on supercomputers. It is based on the SPMD (single program multiple data) paradigm, i.e., after partitioning the problem into smaller pieces, each node in a supercomputer

[1] `https://www.top500.org/lists/2018/11/`

uses the same code to treat its local part. MPI provides the right abstractions to be performance portable across different supercomputers, i.e., MPI programs are usually written to decompose the problem for an arbitrary number of nodes. Each thread of execution is called an MPI rank, and in traditional MPI applications an MPI rank maps either to a single compute node (and utilizes e.g., OpenMP to leverage multicore CPUs) or to a core.

In this work we use the problem decomposition naturally provided by MPI programs in order to execute them on GPUs: we first compile the MPI code to GPU kernels and map MPI ranks to GPU threads. The abstractions offered by MPI allow us to do this transparently, i.e., if a rank initiates communication with another rank, our MPI library implementation will detect if both ranks run on the same GPU, on different GPUs in the same node, or on different nodes, and use the correct low-level communication primitives.

In this design document we outline a framework to transform applications with minimal modifications to the source code into a version ready to run in CUDA devices. We focus on the NVIDIA Volta [4] architecture due to its widespread availability in supercomputers (compared to AMD GPUs) and strong memory consistency guarantees (compared to NVIDIA GPUs based on older architecture).

2.2 Requirements

The main components of our framework are

- a runtime library which supports a sufficient subset of MPI to allow porting existing MPI applications (we will initially support Send/Recv/Bcast/Reduce/Allreduce and Put/Get) and
- a launcher program to start MPI jobs across different GPUs.

The decision which functions to implement is driven by which programs are used for initial benchmarking, we target LAMMPS and AMG from the Sequoia benchmarks, as well as iPIC3D and Nek5000 from the EPIGRAM-HS applications.

2.2.1 Functional Requirements

Our MPI library will support:

- Blocking and non-blocking two-sided point-to-point operations
- Support one-sided operations
- Support for blocking and non-blocking collective operations

In order to compile HPC applications for GPUs, we also need to account for calls to external libraries, such as libc, which are not supported on GPUs. Supporting all of libc is out of scope of this project, we will only support the subset of libc (and other external libraries) which are used by our target applications and benchmarks. This includes functions such as `printf()` and `gettimeofday()`.

The MPI interfaces we will implement are are specified in the MPI Standard [5]. The most up-to-date definitions for persistent collective operations are contained "2018 Draft" version of the MPI Standard[2].

2.2.2 Non-Functional Requirements

To compile MPI programs for GPUs using the NVIDIA compiler, all functions need to be prefixed with a `device` attribute. Global variables need to be to be prefixed with aprivatized and the original main function needs to be replaced by a function with a name recognized by the launcher. In a first prototype we rely on the user to manually perform these changes. However, to improve the user experience it is possible to develop a compiler pass or a source-to-source compiler which performs these steps. We consider this a non-functional requirement.

2.2.3 Applications and Use Cases

We will evaluate the performance of our MPI implementation using a set of microbenchmarks and applications. Of the EPIGRAM-HS applications, Nek5000 is a good candidate because it is a large code base and a rewrite of Nek5000 for GPUs is not possible due to its size.

2.3 Architecture

We first describe the NVIDIA Volta architecture, then continue to describe how we launch GPU-MPI applications and finish with a description of the implementation of collectives and point-to-point operations in GPU-MPI.

2.4 NVIDIA Volta Architecture

The NVIDIA Volta architecture brings a series of new (compared to previously released NVIDIA GPUs) features which facilitate the communication between threads in the GPU, allowing memory to be shared efficiently between blocks, and even between devices on the same bus. This section presents the overview of processing in NVIDIA GPUs, and some brief discussion of the new V100 features. GPUs are essentially SIMD machines, which offer massive parallelism to speed up computations. In these devices, threads are grouped together into *warps*, and at a given moment one warp will be present in one of the processors. If all threads in a warp are executing the same instruction, then they are all being executed at the same time. However, if some of them are in different branches, then they will be executed in sequence, therefore thread divergence should be avoided in order to achieve high performance.

The grouping of threads in warps is performed by the device and the user has no control of which threads fall in which warps. However, another granularity of grouping is *thread blocks*. These are groups of threads which are defined explicitly by the user when the GPU processing begins. As such, the user can finely control which threads are in which block. On the device, a warp is always defined as a subgroup of a block, which

[2] `https://www.mpi-forum.org/docs/drafts/mpi-2018-draft-report.pdf`

provides some guideline in creating threads which can be effectively run in lockstep. Warp sizes are much smaller than the maximum number of threads in a block.

The GPU processors, also known as streaming processors (SM), have one or more blocks present. A block will only run in a single processor at a time, its threads will never be spread among other SMs. Historically, once a block starts running in a SM it was never preempted, and it would run entirely before another block could be scheduled. This has changed in newer architectures, and Volta in particular allows preemption of blocks.

Finally, the blocks are grouped in a kernel, which is the top level group in GPU processing. A kernel is effectively a GPU program, and much of the processing is defined by its life-cycle, with much of synchronizations and memory transfers being defined as valid before it starts and after it ends. In older architectures, a kernel has affinity with a single device, and some devices will only allow one kernel to run at a time. In contrast, Volta allows multiple kernels to run at once.

The memory inside a V100 GPU is organized as a Unified Virtual Address Space. In practice, the memory is shared between the GPU and CPU which controls the GPU kernels. The sharing, however, has several constraints, especially in regard to visibility. Memory is only visible between the GPU and the CPU on explicit synchronization, e.g., using atomics, or implicit synchronization points such as a kernel start or end.

An interesting new feature for newer NVIDIAs architectures is the Cooperative Groups. A cooperative group is a kernel launched with a special call, it behaves similarly to a regular kernel, but it gives access to the thread group primitives. A thread group is a partition of a block in smaller groups, usually defined by the GPU. The main goal is to allow the synchronization of threads on that particular group, without requiring the entire block to synchronize. In practice, this allows memory synchronization between blocks.

Currently the following optional parameters, specific to GPU-MPI launching are available:

- `threads-per-block` specifies the number of threads per block
- `blocks-per-device` specifies the number of threads per device

If the number of ranks requested by the user is larger than what fits in a single device, then they are automatically distributed to other devices, if they are available.

2.4.1 Collectives

MPI collectives are currently only supported on the global communicator. The implementation uses Cooperative Groups to perform a barrier between the ranks and thus synchronize the memory.

As an example we show the pseudocode for a Gather:

```
Gather(sendbuf, recvbuf, size, root)
  // buffer is preallocated
  buffer[myrank] = sendbuf
  Barrier() //to ensure visibility
  if (myrank == root) memcpy(recvbuf, buffer)
```

Note that we require a preallocated communication buffer. This communication buffer is allocated during MPI initialization. In case we run out of preallocated buffer space the communication is performed in multiple rounds, using a fixed amount of memory per round. We also plan to support Planned Collectives, as described in Section 3. This will allow us to allocate sufficient memory in the initialization phase of the collective and thus to always choose the fast-path (using a single memory-barrier) during collective execution.

2.4.2 Point-to-Point

The communication between ranks in a single device, which involves memory being copied, is done by a copy kernel (CK). This standalone application runs in parallel with the main kernel, using free SMs in the GPU to copy data between memory locations. It is launched separately and remains idle until data is available to be copied, then it performs the copy and returns to the waiting state.

Each block on the CK waits on a lockless queue, waiting for data. When a copy starts, the main kernel will fetch a preallocated buffer in memory, and will add it to the CK block 0 queue, in a lock-free fashion. The thread 0 in this block will then fetch the data, and forward it to two other blocks. The process repeats until all blocks have the data buffer. After forwarding the data, the thread 0 will make it available for every thread in the same block, and each will start to copy a segment of the memory. After all threads are done, thread 0 will resume its wait on the associated queue. When the copy is finished, each block atomically updates a counter in the memory buffer, which is used later by the main kernel to query whether the copy is finished or not.

The CK is launched as a single kernel at the same time as the main application, on a different stream, with a fixed number of blocks and threads. It is important to notice that proper care must be taken when launching it, since it must fit the GPU at the same time as the main kernel. When the main kernel calls the MPI Finalize to finish execution, a special command is queued in the CK blocks. This command will signalize the thread the execution has finished, and the kernel may terminate. Each block will finish pending operations and then return, until the kernel finishes executing. Communication across hosts uses the transport kernel (TK) for memory synchronization and MPI to exchange messages across nodes.

2.4.3 Launcher

GPU-MPI Applications are submitted by using our launcher, `rv1run`. Its semantics are similar to `mpirun`, where the user specifies the number of ranks and the hosts where the application will run. The host semantic here refers to the nodes where GPU devices are present. The distribution of the ranks over GPU devices is currently done automatically, since some constraints exist on the devices. It is possible, using flags specific to our launcher to exert more control on where the ranks will run.

As we use cooperative kernels, the number of threads on each device must be the same. Thus, the maximum of number of ranks is restricted by the capacity of the smallest device.

Furthermore, we reserve blocks for support kernels, which further limits the capacity.

The launcher itself is a light wrapper around mpirun, since the inter-host communication relies on MPI, the application is relaunched using mpirun.

2.5 Interfaces

The interfaces implemented within the scope of this project are a subset of those defined by the MPI standard [5]. Specifically, we plan to implement the functions defined in Sections 3, 5 and 11 of the standard. In our first prototype we omit support for Derived Datatypes (we only allow basic types) and Communicators (all operations use the global communicator).

2.6 Technical Terms and Definitions

- **GPU:** A graphics processing unit (GPU) is a specialized electronic device, designed to accelerate the creation of images in a frame buffer intended for output to a display device. In recent years GPUs have become widespread in high-performance computing, because their architecture allows them to perform a certain class of computations faster and more energy efficient than traditional processors (CPUs).
- **MPI:** The Message Passing Interface, a commonly used framework to parallelize scientific applications across many compute nodes.
- **Launcher:** A small program used to start the main application, which itself is a standalone executable, on may nodes in parallel.

3 MPI planned collectives on heterogeneous systems

3.1 Objective

Achieving optimal performance for applications on modern HPC machines is challenging. The trend toward specialized devices increases the hardware heterogeneity and consequently makes optimal performance even harder to achieve. In the realm of network communication, MPI is the *de facto* standard expression of the message-passing programming model and its use is ubiquitous throughout the HPC community. Optimizing MPI communication therefore has a direct impact on HPC outcomes.

Almost all HPC applications rely on MPI for performance-critical communication, even on traditional homogeneous systems. As systems become more complex and more heterogeneous, applications will continue to rely on MPI to provide a high-performance abstraction for message-passing communication. Although alternatives to MPI are effective for some applications in some circumstances, the majority of HPC usage depends on MPI. The current API provided by MPI is powerful and expressive; library implementations are typically portable and of sufficient quality and performance. However, even the most recent MPI Standard has limited support for heterogeneous systems. Therefore, MPI must urgently be improved so that it can continue to play its pivotal role in the HPC ecosystem.

The current MPI Standard provides a rich interface, with several ways to achieve similar or identical outcomes. The choice of functionality is non-trivial but, in general, collective communication should be used by applications in preference to point-to-point communication (or one-sided communication) unless the communication pattern changes very frequently or is not known by all involved processes in advance. Collective operations are easier to understand and use in applications, or via application frameworks, than directly using MPI RMA functionality. Note that any sequence of point-to-point operations can be expressed using a single collective operation, such as a single neighbourhood collective operation in place of the many individual point-to-point sends and receives that achieve the ubiquitous halo-exchange communication pattern. The widespread usage of MPI collective operations (both reported now and expected at Exascale) was demonstrated in the recent survey of MPI usage in the DOE Exascale Computing Project (ECP) [6, 7]

Persistent collective operations are important for Exascale application programming and heterogeneous systems because they allow for long-running, expensive optimisations of complex communication patterns and permit the high costs of those optimisations to be amortised over many usages of the planned operation. Figuring out exactly the best way to organise and utilise heterogeneous hardware to give optimal performance for complex communication patterns without disrupting ongoing computation is still a hot research topic. However, several good options are worth implementing and testing in the short-term, while more advanced and harder-to-achieve options are being fully investigated.

The MPI Forum recently approved the introduction of persistent collective communication operations into the MPI Standard. These new operations will be available from MPI version 4.0, which is due to be ratified and released in Q4 2020. Work has already begun to make use of this new MPI interface — a reference implementation was created

to support the standardisation procedure [8, 9]. The reference implementation demonstrates how to integrate an optimisation, which was suggested by the EPiGRAM project (a precursor to this project), into the Open MPI library and showcases the resulting performance improvements. This integration was not done previously because the suggested optimisation required a planning step, which was not available in MPI before persistent collectives were accepted into the MPI Standard.

It is now urgently required that additional optimisations are researched and implemented successfully to take full advantage of this new interface to improve application performance on heterogeneous systems.

Initial work will focus on building collective communication using one-sided communication primitives (rather than the point-to-point primitives currently used). This implementation approach could be built as a separate library on top of current MPI, or as an enhancement to an existing MPI library. It can immediately take better advantage of modern RDMA hardware and is easier to apply to heterogeneous hardware because it decouples movement of message data from the synchronisation of processes or threads. This project will leverage the Open MPI library by creating a new module its Module Component Architecture (MCA), called libPNBC. Technically, this works involves augmenting the internal schedule-creation mechanism to recognise and execute one-sided commands, e.g. put, get, accumulate, lock and unlock, and designing new low-level schedules using these new commands that minimise the involvement of the CPU in communication tasks. This project will also investigate ways to create and initialise MPI Window memory without blocking execution of the calling application by synchronising all involved processes. This permits additional overlap opportunities and reduces the disruption to ongoing computation (it may also lead to additional standardisation effort in future, but that would be a side benefit only).

Additional work in this work package is planned for work on offloading communication operations (using the persistent collective interface to hide costs) to FPGAs and smart NICs, including the Mellanox SHARP technology. Technically, this work involves programming the internal command schedule into the smart NIC or FPGA hardware, and triggering the new capability on demand in response to application calls to MPI interface functions. In theory, this reduces the overhead of communication to near-zero and maximises the CPU/GPU time available for computation.

The relevant project objective is:

- Implement efficient high-level communication patterns for distributed heterogeneous systems.

The goals required to achieve this objective are:

1. to explore the range of planning optimisations now available via persistent collective operations

2. to improve Open MPI by implementing planning optimisations for key persistent collective operations

3. to measure the impact of planning optimisations in terms of performance improvements for applications

4. to specify new MPI interfaces that expose internal implementation options, where appropriate and useful

3.2 Requirements

3.2.1 Functional Requirements

The functional requirements for network-related capabilities in the EPiGRAM-HS applications are fixed by the communication patterns needed for their chosen computational algorithms. In particular, these applications require the halo-exchange communication pattern for domain-decomposition computational algorithms, transposition within a sub-group of processes for spectral-based computational algorithms, and global reduction for back-propagation Deep Learning algorithms. All of these communication patterns can be expressed using MPI collective communication operations, specifically halo-exchange can be expressed using a neighbourhood all-to-all collective operation (e.g. `MPI_NEIGHBOR_ALLTOALL`), transpose can be expressed using a vector all-to-all collective operation (e.g. `MPI_ALLTOALLV`), and global reduction can be expressed using an all-reduce collective operation (`MPI_ALLREDUCE`).

The functional requirements for all existing MPI functions are specified in the MPI Standard. The most up-to-date definitions for persistent collective operations are contained in the "2018 Draft" version of the MPI Standard[3].

3.2.2 Non-Functional Requirements

Typically, applications express the halo-exchange communication pattern using MPI point-to-point functions. Any sequence of independent point-to-point functions can be equivalently expressed using a single neighbourhood collective function. There is no functional requirement that determines which of these options should be chosen because the functionality is equivalent. However, as collective operations provide more opportunity for MPI to optimise than the constituent point-to-point operations, typically, there are non-functional reasons (performance improvements due to combining function calls and overlap of communication operations with each other) to favour collective operations. Similarly, nonblocking operations should be favoured over blocking operations because they afford the opportunity for overlap with computation, and persistent operations should be favoured over nonblocking operations because they afford the opportunity for amortisation of setup costs.

The non-functional requirements are the main driver for this work. For this work package, these requirements are formalised as follows:

- to improve the communication performance of key MPI persistent collective operations in Open MPI relative to the performance of existing blocking and nonblocking implementations of those operations in Open MPI.

- to improve the overlap of MPI communication with computation by reducing the CPU time spent initiating and completing MPI communication operations.

[3] `https://www.mpi-forum.org/docs/drafts/mpi-2018-draft-report.pdf`

- to improve the overall performance of applications by replacing blocking and non-blocking collective communication MPI operations with optimised persistent collective communication MPI operations.

3.2.3 Applications and Use Cases

The IFS application is already structured to perform one-off setup tasks and can, therefore, easily incorporate the planning step for persistent collective operations. The key MPI operation for this application is:

- `MPI_ALLTOALLV_INIT` using sub-communicators to define which MPI processes are communicating.

The Nek5000 application performs a complex gather-scatter communication pattern consisting of three stages: nearest neighbour exchange, message aggregation, and scatter collective. The key MPI operation for this application is:

- `MPI_NEIGHBOR_ALLTOALL_INIT` using virtual topologies to define which MPI processes are communicating.

The iPIC3D application performs two communication tasks: exchange of particle information and exchange of field information. The key MPI operation for this application is:

- `MPI_NEIGHBOR_ALLTOALL_INIT` using virtual topologies to define which MPI processes are communicating.

The Deep Learning applications use global reduction communication for data parallelism and point-to-point communication for model parallelism. It may be possible to express the point-to-point communication in terms of neighbourhood collective communication. The key MPI operations for this type of application are:

- `MPI_ALLREDUCE_INIT` using sub-communicators to define which MPI processes are communicating.

- `MPI_NEIGHBOR_ALLTOALL_INIT` using virtual topologies to define which MPI processes are communicating.

Note that the intention in this project is to focus on using GPI communication, instead of MPI, for the Deep Learning applications.

3.3 Architecture

The Open MPI library already contains a Modular Component Architecture (MCA), depicted in Figure 2. This project will refactor the relationship between persistent collective communication interface and the lower-level communication components to improve implementation efficiency (also depicted in Figure 2). The location of Finepoints/channels/streams in this architecture diagram has not yet been finalised – early prototype work builds on top of Single-sided Communication, but efficiency savings are possible by building instead directly on top of the BTL or MTL components.

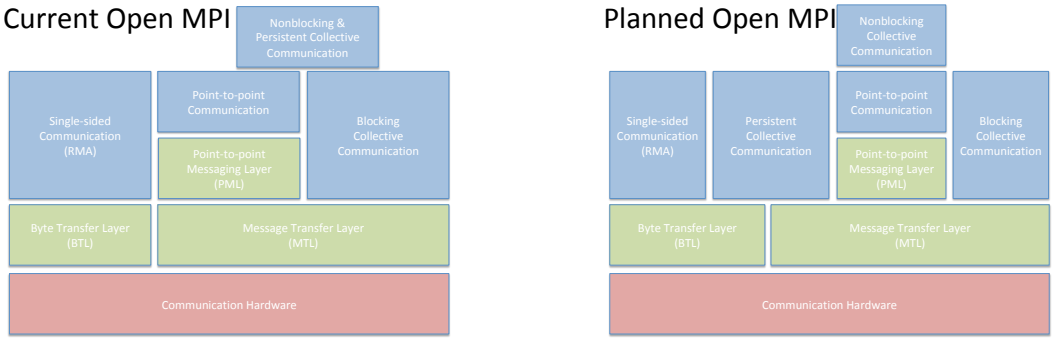

Figure 2: Block diagrams of the current and planned component architecture for Open MPI showing that persistent collective communication is currently built on top of non-blocking point-to-point communication but will be built directly on lower-level communication components to improve middleware efficiency.

3.4 Planned Features

Specific tasks within the technical plan for this work include:

- Creation of a new component for the Modular Component Architecture (MCA) in the Open MPI library, which implements persistent collective operations using MPI RMA operations. This will be constructed in a way that could be re-purposed as a separate library on top of any MPI library to achieve greater portability.
- Improvements to the Mellanox SHARP technology (in collaboration with Mellanox) to support offload of persistent collective operations in scenarios that improvement performance and overlap for EPiGRAM-HS applications.
- Investigation (in collaboration with the University of Tennessee Chattanooga) of the available design-space of mechanisms for notification and state-change within persistent collective step-wise schedules and plans, including triggered operations (hardware and firmware only), interrupts (hardware to software interface), and events (software only).

All of these technical tasks and research collaborations have begun initial work but have not yet progressed far enough to report results. Future deliverables will provide detailed information about these activities and their impact on application performance.

3.5 Interfaces

Interface 1: MPI persistent collective operations

- **Status:** *existing (no changes).*
- **Synopsis:** *Planned collective communication, as defined for MPI-4.*
- **Details:** *Refer to the MPI Standard 2018 Draft specification for full details.*

- **Example/Usage:** *Key functions of interest for this project are:*
  - `MPI_ALLREDUCE_INIT` using sub-communicators to define which MPI processes are communicating.
  - `MPI_ALLTOALLV_INIT` using sub-communicators to define which MPI processes are communicating.
  - `MPI_NEIGHBOR_ALLTOALL_INIT` using virtual topologies to define which MPI processes are communicating.

Interface 2: MPI partitioned communication (aka Finepoints, channels, or streams)

- **Status:** *new (tentative)*
- **Synopsis:** *Early proposals for new MPI semantics related to multi-threading support*
- **Details:** *Refer to the MPI Forum proposal #136*[4] *and the accompanying research paper*[5] *for full details.*
- **Example/Usage:** *the currently planned usage of the proposed interface follows the pattern shown in this pseudo-code:*

```
MPI_Psend_init(numPieces, request)
LOOP (many times)
    MPI_Start(request)
    #pragma OMP PARALLEL (numThreads = numPieces)
    {    MPI_Pready(pieceNum, request)    }
    MPI_Wait(request)
END LOOP
MPI_Request_free(request)
```

3.6 Technical Terms and Definitions

There is a proposal to clarify the specification in the MPI Standard for some of the fundamental MPI terms and conventions. The new definitions for terms that are used in this document are briefly summarised below, but the reader should reference the proposal[6] and the associated research paper [10] for a full explanation of these terms.

- **Persistence/persistent operation:** an MPI operation that consists of 4 stages, each with its own MPI function: initialisation, starting, completion, and freeing.
- **Collective operation:** an MPI operation that involves a group of MPI processes.
- **Blocking operation:** an MPI operation that is performed by a single MPI function call, which blocks until the operation is locally complete.

[4] `https://github.com/mpi-forum/mpi-issues/issues/136`
[5] `https://2019.isc-program.com/presentation/?id=pap147&sess=sess227`
[6] `https://github.com/mpi-forum/mpi-issues/issues/96`

- **Nonblocking operation:** an MPI operation that is performed by two MPI functions: one that initiates the operation and returns before the operation is finished, and another that completes the operation.
- **Finepoints:** a previous name, which has been superseded by "partitioned MPI communication". Refer to the MPI proposal for full details[7].
- **Overlap:** Concurrent execution using parallel hardware to reduce wall-clock time relative to sequential execution.

4 GPI for distributed FPGAs

4.1 Objective

Field Programmable Gate Arrays (FPGAs) offer a different perspective on accelerating applications with hardware. With GPU and other instruction based architectures, application developers need to adapt the algorithm to the hardware, while FPGAs can be tailored to the algorithm. Despite clear benefits of using FPGAs such as low latency, potentially a high bandwidth and for many applications a good energy efficiency, few application developers are using FPGAs due to the difficulty to program them. This has been one of the biggest hurdles for the integration of FPGAs in HPC architectures. High Level Synthesis (HLS) tools start to become mature, making FPGAs easier to program. Currently there is no notable MPI nor GPI support for FPGAs. The objective is to extend the GPI implementation to support distributed FPGAs beyond a simple co-existence of the network stack and the FPGA accelerator by exposing FPGA resources and allowing a direct data transfer. In the scope of EPiGRAM-HS we will add GPI support for distributed FPGAs with communication aided by the CPU.

When designing a communication API for distributed FPGAs, we are first looking into the different options to attach a FPGA to a CPU. Three options are depicted in Fig. 3. In distributed scenarios FPGAs are typically connected to CPUs (as shown in Fig. 3 (a) and (b)). The first option is to incorporate FPGAs onto the same board as the CPU with a tight and connected memory coupling between the two devices. The second option (and currently the most popular one) is to implement the FPGA on a daughter-card and communicate with the CPU over a high-speed point-to-point interconnect such as PCI-e. A third option is to deploy FPGAs by hooking the FPGA directly into the network. Joining a NIC to the FPGA enables the FPGA to communicate directly without any support from a host. In the scope of EPiGRAM-HS we choose a light-weight approach to support communication of distributed FPGAs which are attached to a CPU with a tight and connected memory coupling as well as FPGAs connected to a CPU over a high-speed point-to-point interconnect. The work in EPiGRAM-HS will complement the GPI contribution to the EuroExa project (described in D4.1) which is based on hardware IP to implement GASPI directly on FPGAs. The EuroExa GPI contribution can be used for distributed FPGAs which might not even be attached to a CPU.

[7] `https://github.com/mpi-forum/mpi-issues/issues/136`

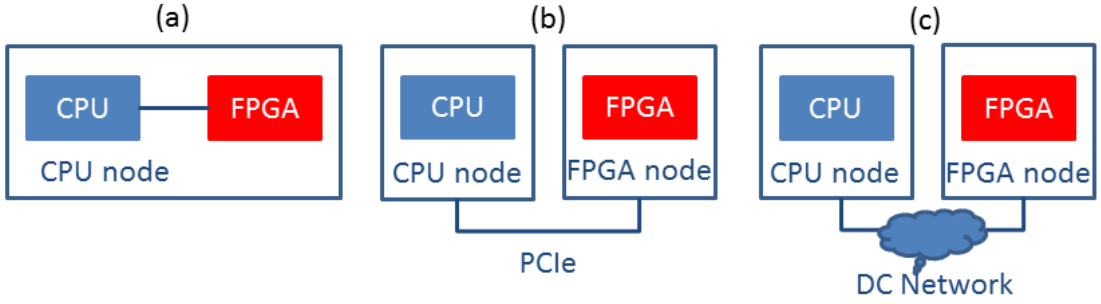

Figure 3: Options for attaching an FPGA to a CPU, (a) onto the same board as the CPU with a tight and connected memory coupling, (b) on a daughter-card to communicate via CPU over a high-speed point-to-point interconnect, (c) by hooking the FPGA directly into the network [11].

4.2 Requirements

4.2.1 Functional Requirements

In this section we give an overview over the additional requirements that GPI needs to fulfill to provide support for distributed FPGAs.

Integration of FPGA memory In order to efficiently utilize the capabilities of distributed FPGAs the application developer has to be aware of the underlying hardware architecture in order to integrate the FPGA memory in the partitioned global address space. GASPI provides configurable RDMA memory segments on which application developers can map the memory heterogeneity. In a tightly coupled scenario the FPGA and the CPU share some of their memory. This memory area can be used as a GASPI segment. If the FPGA is connected over a high-speed point-to-point connection an extra data transfer needs to be handled.

Synchronization of the data Transfer GASPI is based on single-sided asynchronous RDMA communication. This allows a direct access to data in remote memories. Thus there is no implicit synchronization. In order to allow for synchronization notifications are being used. In addition to the communication request, the sender also posts a notification. The order of the communication request and the notification is guaranteed to be maintained. In case of a tightly connected shared memory between the CPU and the FPGA (i.e. Fig. 3 (a)) the current notification mechanism of GASPI is the only mean which needs to be taken into account. However in case that an additional data copy is necessary, i.e. the FPGA is connected to the CPU via a point-to-point interconnect as depicted Fig. 3 (b), the additional transfer between the CPU and the FPGA can be triggered after the notification has been received.

4.2.2 Non-Functional Requirements

Zero-copy data transfers In order to implement a low-latency approach we aim to provide zero-copy data transfers if possible. Zero-copy is defined to be the capability to transmit data out of a user-space buffer via the network into a different user-space buffer without creating another copy of the data in a different memory buffer. Zero-copy transfers allow for lower latency (since the time for additional copies is saved), do not pollute the CPU cache unnecessarily and do not spend CPU cycles on copying data. In a setup where the FPGA and the CPU share memory (i.e. Fig. 3 (a)), a zero-copy approach is feasible by creating a GASPI segment in the common memory area. However in case that the FPGA is connected to the CPU via a point-to-point interconnect (i.e. Fig. 3 (b)) zero-copy can not be supported. If using the pinned memory a copy process from the CPU to the FPGA can be triggered easily. If using the DMA the CPU is not involved in the transfer, i.e. not spending CPU cycles on copying data.

4.2.3 Applications and Use Cases

In the past few years there has been a rise in interest using FPGAs as accelerators for HPC applications. An indication is that a number of EC-funded HPC projects such as EuroExa [8], ExaNoDe [9] and Ecoscale [10] are using FPGAs as their main technology. The projects explore a number of use cases from seismic exploration to astrophysics. In the context of EPiGRAM-HS deep learning applications are relevant.

Deep learning applications are heavily relying on accelerators during execution, because the computation patterns are highly homogeneous, i.e. the most common operations are multiply-accumulate operations. GPUs are nowadays quasi-standard deep learning accelerators. Although GPUs are very fast, they are not as power efficient as one could be for the given operations. This has triggered the emergence of highly efficient specialised hardware including FPGAs for the deep learning domain (as described in deliverable D5.1).

Current deep learning networks are too big to be implemented on one FPGA only. Therefore efficient communication between multiple FPGAs is key to a successful implementation of deep learning applications on FPGAs. We do not yet know how deep learning applications will make use of FPGAs. I.e. if clusters of FPGAs directly hooked into the network as sketched in Fig. 3(c) will be on the rise or whether FPGAs will be attached to CPUs (Fig. 3(a) and (b)) as it is often currently the case. In the scope of this project we concentrate on the case of FGPAs which are attached to CPUs and improve the support of inter-node communication APIs for such a model.

The EPiGRAM-HS project explores possible directions of deep learning framework developments in WP4 and WP5. The proposed design is connected to the deep learning extensions of the task-based workflow management system GPI-Space for parallel applications which can use GPI as basis of its independent memory layer and a direct usage

[8] https://euroexa.eu/
[9] http://exanode.eu/
[10] http://www.ecoscale.eu/

with GPI. In WP4 GPI-Space will be extended to FPGA-enabled applications. While the work in WP4 is focussed on allowing multiple task implementations and the support for dynamic system reconfigurations, the work in WP2 tackles a seamless integration of the communication to the FPGAs connected to a CPU. We will start to test the deep learning applications on FPGAs using deep learning computation kernels developed for FPGAs which will be statically distributed.

The GPI implementation to support communication between distributed FPGAs will enable the Deep Learning applications of EPiGRAM-HS namely the lung cancer detection and malware classification applications.

4.3 Architecture

The GASPI [1] standard, for which GPI [2] is the only implementation, leverages remote completion and one-sided RDMA driven communication in a Partitioned Global Address Space. The global memory can be accessed by other nodes and is divided in contiguous blocks of virtual memory called segments. One-sided asynchronous communication is the basic communication mechanism. The entire communication is managed by the local process only. The remote process is not involved. In general communication with the GASPI interface has different phases:

- As a result of an **initialization phase** the resources have been allocated and meta data has been exchanged. This phase is typically executed once at the beginning of the program.
- At the end of the **setup communication phase** the target buffer has been explicitly synchronized and validated. The setup communication phase is a prerequisite for the start of communication. This phase is not needed if double buffering with symmetric communication patterns are being used, because the information is implicitly available.
- A **communication phase** starts as soon as the source buffer is filled.
- As a result of the **end of communication phase** the data transfer completion is explicitly acknowledged to assure that the source buffer can be overwritten. This phase is not needed if double buffering with symmetric communication patterns are being used.
- A **final phase** is needed to release the resources. This phase is typically executed at the end of the program.

As summarized in Table 1 notifications are being used for the setup communication phase, the communication phase and the end of communication phase. Notifications allow for synchronization between distributed nodes. Notifications can be used standalone (`gaspi_notify` command) or in conjunction with a data transfer (e.g. `gaspi_write_notify` in case of a write). The notification semantic is complemented with routines which wait for an update of a single or even an entire set of notifications on the remote side (`gaspi_wait_some` waits for a set of notifications).

phase	producer	description	consumer	description
init	allocate resources, exchange meta info	setup phase	allocate resources, exchange meta info	setup phase
setup communication	`wait_some`	check before comm.	`notify`	start as soon as receive buffer can be changed
commmunication	`write_notify`	start as soon as source buffer is filled	`wait_some`	check before work on receive buffer
end of communication	`wait_some`	check if source buffer can be overwritten	`notify`	send as soon as data data arrived
final	release resources	final phase	release resources	final phase

Table 1: Communication phases in GASPI

Figure 4 outlines the communication pattern in GASPI. The consumer indicates the target buffer validity before the write begins. Then the producer commences the data transfer and issues a notification. On the receiving side GASPI guarantees that data is locally available whenever the corresponding notification becomes locally visible. This mechanism enables fine-grain (request based) asynchronous dataflow implementations. Once data has arrived the consumer acknowledges completion of the data transfer, so the producer can reuse the buffer.

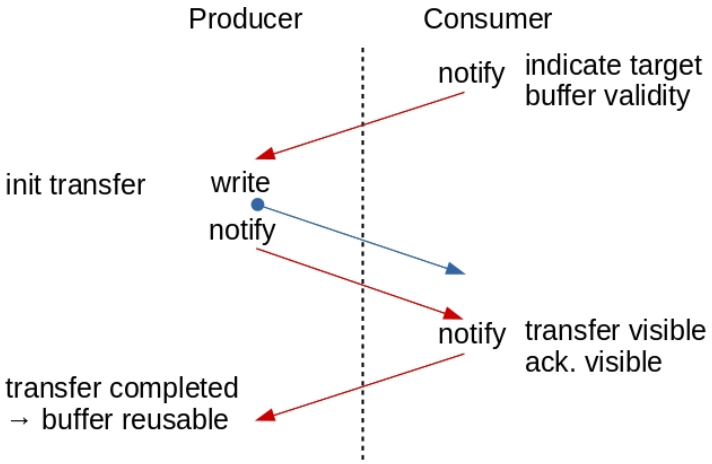

Figure 4: Interface in GASPI with `gaspi_write`

This generic approach will be customized to support distributed FPGAs. The communication sequence is depicted in Fig. 5 for embedded FPGAs (i.e. FPGAs sharing

memory with the CPU). In our model synchronization is still handled by the CPUs, the data transfer is extended to work with an attached FPGA. Fig. 5 shows a minimal diagram which is equivalent to the communication and a subsequent computation step.

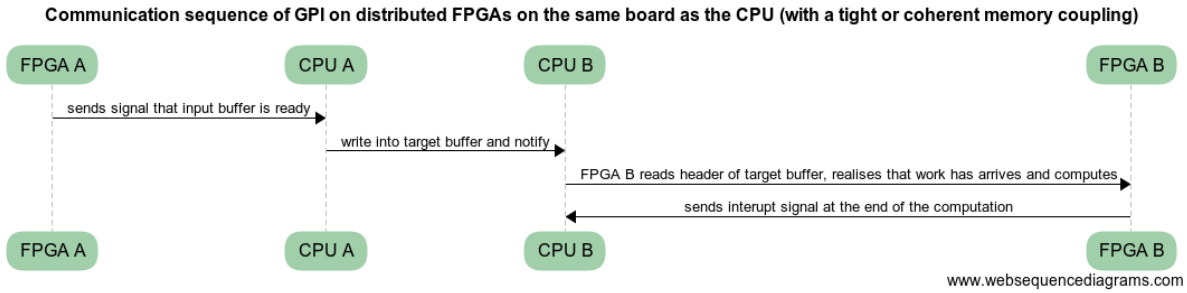

Figure 5: Minimal sequence diagram for CPU aided GPI communication for distributed embedded FPGAs

4.4 Interfaces

Interface: GASPI Segment

- **Status:** existing (no changes)
- **Synopsis:** Interface for application developers in order to allocate and register contiguous memory areas for usage in the global partitioned address space of GASPI
- **Details:** Special care needs to be taken by the application developer to allocate a memory area which is either tightly connected between the CPU and FPGA or pinned in case that the FPGA is connected to the FPGA.
- **Example/Usage:** The interface can be found in the GASPI specification [1]. Several methods are existing.

Interface: Synchronization of the communication phase

- **Status:** existing (to be changed)
- **Synopsis:** The current interface uses notifications for weak synchronization which are sent by the producer and checked by the consumer in a two stage process using `gaspi_notify_waitsome` to check that the notification with correct id has arrived and `gaspi_notify_reset` to check the notification value to ensure thread safety. `gaspi_notify_reset` resets the notification with ID `notification_id` to zero. The function is an atomic operation: Threads can use it to safely extract the value of a specific notification.
- **Details:** `gaspi_notify_waitsome` and `gaspi_notify_reset` already exist. In order to simplify the synchronization in case that a copy from the GASPI segment on a pinned memory allocation on the CPU to the FPGA memory is needed, the `gaspi_notify_reset` command can be overloaded to handle the copy process from CPU to FPGA.

- **Example/Usage:** The interface can be found in the GASPI specification [1]. Especially interesting for the purpose of the project is the `gaspi_notify_reset` function.

4.5 Technical Terms and Definitions

- **Segments:** The global memory can be accessed by other nodes using the GASPI API and is divided into so-called segments. A segment is a contiguous block of virtual memory. The segments may be globally accessible from every thread of every GASPI process and represent the partitions of the global address space. The segments can be accessed as global, common memory, whether local - by means of regular memory operations - or remote - by means of the communication routines of GASPI. Memory addresses within the global partitioned address space are specified by the triple consisting of the rank, segment identifier and the offset.

- **Communication queues:** GASPI provides the concept of message queues. These queues facilitate higher scalability and can be used as channels for different types of requests where similar types of requests are queued and then get synchronized together but independently from the other ones (separation of concerns, e. g. one queue for operations on data and another queue for operations on meta data). The several message queues guarantee fair communication, i. e. no queue should see its communication requests delayed indefinitely

- **One-sided communication:** One-sided asynchronous communication is the basic communication mechanism provided by GASPI. The one-sided communication comes in two flavors. There are read and write operations from and into the partitioned global address space. One-sided operations are non-blocking and asynchronous, allowing the program to continue its execution along the data transfer. The entire communication is managed by the local process only. The remote process is not involved. The advantage is that there is no inherent synchronization between the local and the remote process in every communication request. At some point the remote process needs knowledge about data availability managed by weak synchronization primitives.

- **Notifications:** In order to allow for synchronization notifications are being used. In addition to the communication request, in which the sender posts a put request for transferring data from a local segment into a remote segment, the sender also posts a notification with a given value to a given queue. The order of the communication request and the notification is guaranteed to be maintained. The notification semantic is complemented with routines which wait for an update of a single or even an entire set of notifications. In order to manage these notifications in a thread-safe manner GASPI provides a thread safe atomic function to reset local notification with a given identifier. The atomic function returns the value of the notification before reset. The notification procedures are one-sided and only involve the local process.

5 Conclusion

The deliverable describes the design of three extensions to the MPI and GPI programming model to support heterogeneity for HPC systems. The proposed extensions aim to:

- **support MPI-RMA for GPUs.** The Volta NVidia GPU architecture, which facilitates the communication between threads in the GPU and allows memory to be shared between blocks and devices on the same bus, will be the target of the developments. In the scope of the project a runtime library which supports a subset of the MPI specification and a launcher program to start MPI jobs across different GPUs will be implemented. The aim is an extension to the MPI implementation which is almost transparent to the user.

- **implement persistent collectives in MPI.** The purpose of persistent collectives is to allow the MPI library to plan its future usage of system resources. The aim is to improve the communication performance of key MPI persistent collective operations and to improve the overlap of computation and computation. The collectives will be implemented on a set of key MPI operations. Persistent collectives can be used to offload the computation for collectives on a network device.

- **add GPI support for distributed FPGAs.** The GPI support for distributed FPGAs will be light-weight with communication aided by the CPU. We will mainly support FPGAs which are attached to a CPU with tight and connected memory coupling as well as FPGAs connected to a CPU over a high-speed point-to-point interconnect. Special care needs to be taken to integrate FPGA memory efficiently and to synchronize the data transfer.

As pointed out in each section respectively each extension to the programming models will be validated by at least one EPiGRAM-HS application.

References

[1] Christian Simmendinger, Mirko Rahn, and Daniel Grünewald. *The GASPI API: A Failure Tolerant PGAS API for Asynchronous Dataflow on Heterogeneous Architectures*, pages 17–32. 11 2015.

[2] GPI-2: Programming the next generation of supercomputers. `http://gpi-site.com/`. Accessed: 2019-05-16.

[3] Javier Delgado, Joao Gazolla, Esteban Clua, and S Masoud Sadjadi. A case study on porting scientific applications to GPU/CUDA. *Journal of Computational Interdisciplinary Sciences*, 2(1):3–11, 2011.

[4] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the NVidia Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[5] MPI Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[6] David E. Bernholdt. A Survey of MPI Usage in the DOE Exascale Computing Project. Technical report, ECP, P.O. Box 2008, Oak Ridge, TN, 37831, 2018.

[7] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E. Grant, Thomas Naughton, Howard P. Pritchard, Martin Schulz, Geoffroy R. Vallee. A Survey of MPI Usage in the U. S. Exascale Computing Project. Technical report, ECP, P.O. Box 62, Oak Ridge, TN, 37831, 2018.

[8] Bradley Morgan, Daniel J. Holmes, Anthony Skjellum, Purushotham Bangalore, and Srinivas Sridharan. Planning for performance: Persistent collective operations for MPI. In *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI '17, pages 4:1–4:11, New York, NY, USA, 2017. ACM.

[9] Daniel J. Holmes, Bradley Morgan, Anthony Skjellum, Purushotham V. Bangalore, and Srinivas Sridharan. Planning for performance: Enhancing achievable performance for MPI through persistent collective operations. *Parallel Computing*, 81:32 – 57, 2019.

[10] Daniel J. Holmes Julien Jaeger Guillaume Mercier Claudia Blaas-Schenner Anthony Skjellum Purushotham V. Bangalore, Rolf Rabenseifner. Exposition, clarification, and expansion of MPI semantic terms and conventions. In *Proceedings of the 26th European MPI Users' Group Meeting*, EuroMPI '19, 09 2019. in peer-review.

[11] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling FPGAs in Hyperscale Data Centers. 08 2015.