Efficient Offloading of Parallel Kernels Using MPI_Comm_spawn

Sebastian Rinke, Suraj Prabhakaran, Felix Wolf
German Research School for Simulation Sciences, 52062 Aachen, Germany
RWTH Aachen University, 52062 Aachen, Germany
{*s.rinke, s.prabhakaran, f.wolf*}*@grs-sim.de*

***Abstract*—The integration of accelerators into cluster systems is currently one of the architectural trends in high performance computing. Usually, those accelerators are manycore compute devices which are directly connected to individual cluster nodes via PCI Express. Recent advances of accelerators, however, do not require a host CPU anymore and now even enable their integration as self-contained nodes that are able to MPI-communicate over their own network interface. This approach offers new opportunities for application developers, as compute kernels can now span multiple communicating accelerators to better account for larger MPI-based code regions with the potential for massive node-level parallelism. However, it also raises the question of how to program such an environment. An instance of this novel cluster architecture is the DEEP cluster system currently under development. Based on this hardware concept, we investigate the MPI_Comm_spawn process creation mechanism for offloading MPI-based distributed memory compute kernels onto multiple network-attached accelerators. We identify limitations of MPI_Comm_spawn and present an offloading mechanism which results in only a fraction of the overhead of a pure MPI_Comm_spawn solution.**

***Keywords*-MPI_Comm_spawn; computation offloading; network-attached accelerators; Intel Xeon Phi; DEEP**

I. Introduction

Since the introduction of graphics processing units with the capability to be programmed using general purpose programming languages such as C, accelerators have started to find their way into cluster systems. Originally designed as PCI Express (PCIe) devices, which require a host CPU, and are therefore attached to individual host systems, latest accelerators no longer need a host and can be connected to other cluster nodes via their own network interface. The first example for this is the Intel Xeon Phi-based accelerator developed within the DEEP project [1]. These so-called network-attached accelerators run their own operating system and are able to execute MPI programs. That is, now application developers have the opportunity to extract larger highly parallel code regions and separate them as compute kernels to be offloaded in parallel across the network onto multiple accelerators. Within such a kernel, MPI calls can be used for communication between different accelerators without requiring any interaction with the cluster nodes. This is currently not supported by CUDA and OpenCL. In essence, kernels can be MPI programs which contain code constructs, such as pragmas, specific to the accelerator hardware and MPI calls for communication. As a more technical consequence, the ratio of special purpose accelerators to general purpose cluster nodes can be customized because the maximum number of accelerators available in a cluster does not depend on the number of PCIe slots provided by its nodes. Since accelerators are now independent from cluster nodes, they can also be requested separately in a batch script according to precise application needs. This avoids the scenario where free accelerators are unavailable because the job allocated on their hosts fully occupies the host resources while leaving the local accelerators unused. Similarly, broken accelerators do not affect the availability of cluster nodes and vice versa. However, it is obvious that data transfers over the network between accelerators and cluster nodes suffer from a greater penalty than over local PCIe only. On the other hand, the ability to communicate between accelerators over MPI will reduce the need to communicate frequently between cluster nodes and accelerators, compensating for this penalty. Of course, kernels can run asynchronously so that cluster and accelerator nodes are utilized simultaneously. A use case example for this scenario are classic MPMD-style multiphysics applications where the different physics calculations within the same time step have different scaling characteristics. Furthermore, in this respect, the iPIC3D implicit-moment-method particle-in-cell code [2] for space weather simulations is a good candidate and investigated in-depth within the DEEP project. More details on this can also be found in [1].

An example for a cluster system that embodies the concept of network-attached accelerators is DEEP. This machine will consist of cluster nodes housing Intel Xeon multicore CPUs and accelerators based on Intel Xeon Phi manycore CPUs. Both cluster nodes and accelerators are equipped with their own network interface and are able to communicate via MPI. That is, compute kernels to be offloaded to the accelerators can be MPI programs with corresponding communication calls for data exchange. These capabilities are not supported by current accelerator programming models such as CUDA and OpenCL, which are designed for node-local accelerators and where data exchange between devices cannot be triggered by the kernel running on the device. Thus, the question remains how to program a cluster with network-attached accelerators and, in particular, how to offload MPI-based compute kernels onto the accelerators without introducing yet another programming model and causing major rewrites of existing applications.

To investigate this question, we use the hardware concept of the DEEP cluster as the basis for our considerations, which is still under development at the time of writing and therefore not available yet. However, as can be seen in the following, for this work it is sufficient to know the hardware specifics and capabilities of the final system for developing proper approaches and drawing conclusions. Note that although considering the DEEP cluster as the hardware basis, our solution developed in this paper is not restricted to DEEP and can be used for every future cluster sytem equipped with network-attached accelerators. Focusing on high performance computing, MPI is the most common programming paradigm for distributed memory systems and thus many applications already use it. For this reason, we present an offloading approach based on MPI's process creation mechanism MPI_Comm_spawn. We review the capabilities of MPI's spawn and develop an approach which compensates for the major drawbacks of a pure spawn-based solution. Our experimental results show that this approach only adds a fraction of the overhead compared to the naive spawn-only approach. The remainder of this work is structured as follows: We start with reviewing related work. After an overview of the DEEP cluster and the MPI_Comm_spawn function call, we present different offloading approaches based on MPI_Comm_spawn. Afterwards, we discuss the implementation of our offloading mechanism and compare this experimentally to an MPI_Comm_spawn-only solution. Finally, we conclude the paper and outline future work.

II. Related Work

Several programming models, including CUDA, OpenCL, and OpenACC, support offloading compute kernels with a high degree of parallelism onto node-local accelerators. rCUDA [3] even enables the execution of CUDA kernels on GPUs attached to remote hosts. Here, communication to the remote hosts is performed via native InfiniBand. Not restricted to CUDA kernels and with more emphasis on resource-management aspects, the concept of the dynamic accelerator-cluster architecture [4] allows the assignment of network-attached accelerators to classic cluster nodes while communication between the host and its remote accelerators is implemented using MPI. However, also this approach does not allow compute kernel functions to communicate with other kernel functions that run on different accelerators. That is, all communication from the host to its assigned accelerators as well as between its accelerators is still host controlled.

As stated above, Xeon Phi-based accelerators can execute MPI processes with usual MPI communication. Current large-scale cluster systems, such as Stampede at the Texas Advanced Computing Center (No. 6 in Top500 Jun. 2013), house Xeon Phi-based PCIe cards that are locally attached to individual cluster nodes. Corresponding MPI libraries with special support for Xeon Phi, such as Intel MPI, MPICH, and MVAPICH, provide the software required for MPI communication. However, regardless of a particular MPI library, state-of-the-art usage scenarios foresee to start all MPI processes on host CPUs and accelerators at once. That is, all MPI processes both on host CPUs and accelerators belong to the same initial MPI communicator MPI_COMM_WORLD. As a consequence, host processes and accelerator processes cannot MPI-communicate independently from each other. For instance, collective communication for accelerators only requires an exclusive communicator including only the accelerator processes. To address this, the application programmer has to create two new communicators, one for the host processes and one for the accelerator processes. Finally, all instances of MPI_COMM_WORLD in the original program code, which is executed on the host, have to be replaced with the corresponding new communicator. These code changes are error-prone and do not follow the conventional intuitive way of incrementally changing existing code to support accelerators. That is, instead of modifying only a small fraction of the original program code to be offloaded to accelerators, the application programmer has to touch a larger portion of the code outside the accelerated regions. To compensate for this drawback, accelerator processes could be started dynamically at runtime using MPI_Comm_spawn. Another aspect is the hardware configuration. With node-attached accelerators, host and accelerator processes compete for the host's network interface as all communication is routed through the host. Network-attached accelerators with their own network interface avoid this bottleneck.

A different approach is HeteroMPI [5], which is designed for programming processors running at different speeds that are connected with network links of different speeds. The main idea is to automate the selection of a group of processors that can execute a given algorithm faster than any other group. The selection process is based on a user-defined performance model of the algorithm and the hardware capabilities of the available processors with their different link speeds. The proposed programming interface is an extension of MPI. However, compared to our work, the focus of HeteroMPI is on heterogeneous networks of general purpose processors without accelerators, which are used and administered independently by multiple users.

Kimura and Takemiya [6] spawn MPI processes via MPI_Comm_spawn to distribute a coupled fluid/structure simulation across several machines to account for the different computational requirements of individual simulation components. Specifically, the fluid dynamics runs on a vector-parallel computer, while the structure dynamics is computed on a scalar-parallel computer. This approach is similar to the DEEP concept, however, it uses different machines instead of combining their different hardware capabilities into a single, more tightly coupled system. Also, the spawned program is supposed to run until the complete simulation finishes. In this work, however, we deal with smaller code regions to be offloaded on demand. Another work considers the MPI spawn function for implementing task parallelism [7]. However, the authors note that the spawn call is too coarse-grained as it only allows starting new programs instead of individual

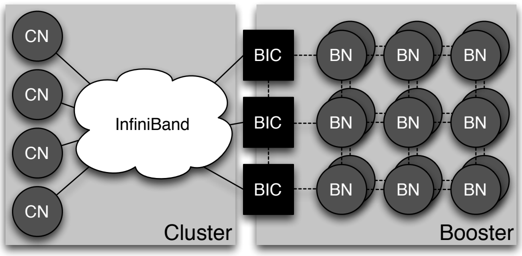

Figure 1. DEEP architecture with cluster nodes (CN), booster nodes (BN), and booster interface cards (BIC). Solid lines are InfiniBand whereas dashed lines are EXTOLL links.

functions. Gangadharappa et al. [8] present an optimized implementation of MPI's spawn call. However, depending on the number of processes spawned, in spite of the optimizations spawn times still range from one to several seconds. Spawning MPI processes at runtime requires support from the resource management system. Current resource managers provide facilities for creating processes during job execution. However, the resources on which the new processes will execute have to be requested before job start, e.g., in a batch script. To enable even more flexible usage scenarios where those resources can be requested and allocated during job execution, [9] presents extensions to the TORQUE/Maui batch system.

III. DEEP Architecture

This section gives an overview of the DEEP cluster system, which is currently under development and will be installed at the Jülich Supercomputing Center, Germany. DEEP implements the concept of network-attached accelerators and will serve as the basis for our considerations on how to program such a system. The DEEP (Dynamical Exascale Entry Platform) architecture consists of two parts: A cluster part and a booster part, see Figure 1. The cluster part will consist of 128 general purpose cluster nodes connected through a QDR InfiniBand fat tree. Each cluster node comprises two Intel Xeon E5-2680 multicore processors (8 cores each). The booster part will contain 512 network-attached accelerators which are connected by a 8 × 8 × 8 3D torus EXTOLL network [10]. Each accelerator, called booster node, houses one Intel Xeon Phi manycore processor (60 cores with a 512-bit vector unit each). The motivation for choosing two different networks and topologies is to better account for the different communication requirements of the main program on the one hand and highly parallel kernels on the other. That is, code parts with a modest degree of parallelism and more irregular communication patterns should run on the multicore cluster part, while highly parallel regions with large thread counts and regular communication patterns should be offloaded to the booster part. To enable communication over the complete machine, that means also between the two parts, so-called booster interface cards are connected to InfiniBand as well as EXTOLL to provide bridging capabilities. In particular, one booster interface card connects 16 booster nodes to the cluster part. A more in-depth description of the DEEP hardware can be found in [1] . Regarding software, the system will provide an MPI implementation that enables transparent communication between cluster and booster. From a user's point of view, an MPI application can be launched over both parts of the system as the cluster nodes and the booster nodes are able to run usual MPI processes. The challenges this approach entails for the application programmer are discussed below.

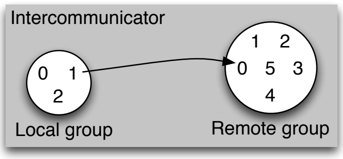

Figure 2. MPI intercommunicator where the arrow denotes communication between two processes. Numbers denote process ranks.

IV. MPI_Comm_spawn

In this section, we discuss MPI_Comm_spawn and related routines which are the basis for our offloading approach. With MPI-2 the dynamic process model has been introduced. That is, MPI allows for the creation of extra processes after an MPI application has started. One way to start new processes at runtime and establish communication between these new processes and the existing MPI application is MPI_Comm_spawn:

```
int MPI_Comm_spawn(char *command, char *argv[],
        int maxprocs, MPI_Info info, int root,
        MPI_Comm comm, MPI_Comm *intercomm,
        int array_of_errcodes[])
```

This call is collective over the intracommunicator `comm`. It tries to start `maxprocs` copies of the MPI program `command` with all spawned child processes getting the same arguments provided in `argv`. Since MPI does not define where the new processes are started, the `info` argument can be used for this and other runtime system specific purposes. The `root` argument gives the rank of the process in `comm` which is the only process where all arguments before the `root` argument are examined. The array `array_of_errcodes` of length `maxprocs` can be used to check for the success of the spawn operation. On success, the newly created processes form their own MPI_COMM_WORLD intracommunicator and an intercommunicator is returned in `intercomm`. This intercommunicator contains two disjoint groups of processes, a local and a remote group. The local group comprises the processes which called MPI_Comm_spawn, i.e., all processes in `comm`. The remote group contains the newly created processes. Note that in both groups process numbering starts with zero. This intercommunicator can be used with, e.g., point-to-point and collective communication, however, as the prefix *inter* suggests, only for communication between not within the process groups. As the child processes do not call MPI_Comm_spawn but MPI_Init once created, they have to obtain their intercommunicator explicitly through the function MPI_Comm_get_parent.

Afterwards, they can communicate with their parents using their acquired intercommunicator. Note that the child processes will find themselves in their local group whereas they will find their parents in their remote group. An example for an intercommunicator is depicted in Figure 2.

V. Offloading Approaches

In the following, we investigate different approaches for offloading computations from cluster nodes to network-attached accelerators. Given that both the main program on the cluster nodes and the compute kernel on the accelerators are MPI programs, it does not seem to be necessary to have a special offloading mechanism at the first glance. One could simply run the MPI application by creating all the processes on cluster nodes and accelerators already at job start. During program execution, each process must then decide which code path to follow based on whether it is running on a cluster node or an accelerator. However, both cluster node and accelerator processes together form one common MPI_COMM_WORLD. As a consequence, for instance, cluster node processes block forever in collective calls on MPI_COMM_WORLD as the accelerator processes would not necessarily want to participate, or vice versa. A solution would be to divide cluster node and accelerator processes into two different communicators and finally replace all occurrences of MPI_COMM_WORLD by the corresponding new communicator. However, this may require major changes in existing applications and is not very practical. One possibility to have cluster node and accelerator processes initially separated in their own MPI_COMM_WORLD is to start the accelerator processes at runtime with MPI_Comm_spawn. This approach is discussed below.

A. Spawn

Using MPI_Comm_spawn for starting the processes on the accelerators gives a separation of cluster node and accelerator processes into two disjoint intracommunicators. This allows for independent communication within each group of processes. Moreover, the resulting intercommunicator can be easily used for communication between both groups as rank numbering starts from zero in each group. Here, the intercommunicator already provides improved semantics over the no-spawn solution from above, as the programmer need not distinguish between ranks from cluster node and accelerator processes. Additionally, collectives for intercommunicators facilitate the data exchange between both process groups and may improve performance compared to using point-to-point primitives. Given the spawn approach for offloading computations, the usage scenario could be the following:

(i) The cluster processes perform one initial spawn to create the accelerator processes.

(ii) They send the input data from the cluster nodes to the accelerators.

(iii) The accelerator processes perform computation on the accelerators, exchanging information between them.

(iv) They send the results back from the accelerators to the cluster processes.

Steps (ii)-(iv) can be repeated on demand.

This approach assumes that the actual calculations on the data remain the same. That is, there is only one type of compute kernel to be offloaded. However, in practice, different compute kernels could be required where even their number of invocations is not known before runtime. To account for this case, the programmer could simply perform a new spawn whenever a compute kernel is offloaded. That is, after finishing any kernel, the corresponding accelerator processes are terminated as well. In this way, a different program (compute kernel) can be provided with every MPI_Comm_spawn. However, the disadvantage is the additional spawning overhead for each kernel invocation as the processes have to be created again. Depending on how a particular MPI implementation represents an MPI process, whether as operating system process or thread, program data could be re-used in subsequent spawn calls. However, most MPI implementations define MPI processes as operating system processes. In this case, the re-creation of processes disables the re-use of data between successive kernel calls, as the processes terminate and thus their data cease to exist. Nevertheless, under the assumption that MPI processes are threads, the spawn call's info argument could be used to implement a more flexible spawn call with less startup overhead and the possibility of re-using the data. However, these assumptions highly depend on the actual MPI implementation used and prevent a portable solution. A portable option is to spawn only once and implement a protocol between cluster node and accelerator processes that notifies the accelerators of which kernel to run. This addresses the limitations of multiple spawns, yet, introduces additional work and pitfalls for the programmer. In the following, we present an approach that offers the advantages of a spawn-once solution and frees the application programmer from implementing such a protocol.

B. Spawn and MPIX_Kernel_call

To take advantage of the one-spawn approach and free the programmer from implementing a protocol for triggering kernel execution on demand, we introduce the function MPIX_Kernel_call. In particular, after the initial MPI_Comm_spawn for creating the accelerator processes, MPIX_Kernel_call is used to start the execution of a compute kernel function whose name is specified as an argument.

```
int MPIX_Kernel_call(char *kernelname,
        int argcount, void *args[], int *argsizes,
        int root, MPI_Comm comm, MPI_Comm intercomm)
```

A call to MPIX_Kernel_call involves exactly the same processes as a previous spawn call did. After spawning, the parents instruct their children to call a function `kernelname`. For this reason, `comm` and `intercomm` are the same communicators as supplied to and returned from a previous spawn call, respectively. Kernel_call is collective over the parents in `comm` and starts the function

```
void main(int argc, char **argv)
{
   // Spawn accelerator processes
   MPI_Comm_spawn(..., comm, &intercomm,
                  ...);

   // Start "kernel0" on accelerators
   MPIX_Kernel_call("kernel0", ...,
                    comm, intercomm);

   // Send input data to kernel functions
   MPI_Alltoall(..., intercomm);

   // Do some other calculations...

   // Recv results from kernel functions
   MPI_Alltoall(..., intercomm);
}
```

Listing 1. Main program launching an MPI kernel.

```
void kernel0(double a, int b, char c)
{
   // Get intercommunicator to parents
   MPI_Comm_get_parent(&intercomm);

   // Recv input data from parents
   MPI_Alltoall(..., intercomm);

   // Do calculations and communicate...

   // Send results to parents
   MPI_Alltoall(..., intercomm);
}
```

Listing 2. MPI kernel launched by main program.

`kernelname` on the children given by `intercomm`. As with usual function calls, it is possible to provide arguments with `args` where `argcount` and `argsizes` denote the number and the sizes of the corresponding arguments. Similar to MPI_Comm_spawn, `root` gives the rank of the parent in `comm` which is the only process where all arguments before the `root` argument are examined. Kernel_call returns once the provided input buffers can be modified again. This does not imply that kernel execution has finished, but tells that the corresponding accelerator processes will be instructed to start kernel execution. The call does not foresee testing for kernel completion. However, a good indicator for completion is when the results have been received on the cluster node. That is, after finishing the calculation, in the last step, the compute kernel can MPI-communicate the results back to the cluster nodes. Similarly, at the beginning, the kernel can receive the input data. Since for many applications using accelerators the availability of the results is sufficient, we did not introduce means for testing for kernel completion. However, more experience with applications which use Kernel_call in the future might change our decision.

According to the description above, MPIX_Kernel_call is complementary to MPI_Comm_spawn. Note that to trigger a function via Kernel_call, this function must be available in the program previously spawned. This and more details are discussed in the implementation in Section VI. Using MPIX_Kernel_call together with MPI_Comm_spawn leads to the following usage scenario:

(i) The cluster processes perform one initial spawn to create the accelerator processes.

(ii) They start a kernel on the accelerators.

(iii) They send the input data from the cluster nodes to the accelerators.

(iv) The accelerator processes perform computation on the accelerators, exchanging information between them.

(v) They send the results back from the accelerators to the cluster.

Steps (ii)-(v) can be repeated on demand. That is, Kernel_call can be executed multiple times, which is why steps (iii) and (v) for moving the data forth and back might not always be necessary. In any case, however, the programmer has to ensure that all messages sent by a kernel can also be received by this kernel running in a different process before starting the next kernel via another Kernel_call. This guarantees that every communication among the accelerators themselves and between accelerators and cluster nodes is finished within the same kernel. Otherwise communication operations of successive kernels might interfere with each other. This is similar to the matching rules in any MPI program, however, here applied at the granularity of a compute kernel instead of a complete program. From a high-level perspective, Kernel_call triggers the execution of the same function on multiple accelerators in parallel. The order of multiple invocations of Kernel_call determines the order in which the requested kernel functions are executed. This approach enables applications to offload compute kernels in any order and any number of times. At the same time, instead of multiple expensive process spawns, each kernel is triggered by a single message sent from the cluster to the accelerator processes. Listings 1 and 2 show an example of how to start a compute kernel.

C. Spawn and MPIX_Kernel_call_multiple

A simple optimization of MPIX_Kernel_call is to instruct the execution of kernels in a batch. That is, instead of sending one message per kernel execution from the cluster to the accelerators, multiple kernel requests can be coalesced into a single message, which further reduces the overhead. The corresponding function is called MPIX_Kernel_call_multiple:

```
int MPIX_Kernel_call_multiple(int count,
     char *array_of_kernelname[],
     int *array_of_argcount, void **array_of_args[],
     int *array_of_argsizes[], int root,
     MPI_Comm comm, MPI_Comm intercomm)
```

It is similar to MPIX_Kernel_call. The only difference is that instead of only one, `count` kernels are requested for execution. The kernels are executed in the order as provided in `array_of_kernelname`. For the i-th kernel, `array_of_argcount[`i`]` contains the number of arguments and `array_of_argsizes[`i`]` provides a pointer to an array with the corresponding argument sizes. The actual arguments for the i-th kernel are accessible via an array of pointers. Finally, `array_of_args[`i`]` stores the pointer to this array. With MPIX_Kernel_call_multiple and MPI_Comm_spawn, the usage scenario becomes:

(i) The cluster processes perform one initial spawn to create the accelerator processes.
(ii) They request the execution of multiple kernels on the accelerators.
(iii) They send the input data from the cluster nodes to the accelerators.
(iv) The accelerator processes perform computation on the accelerators, exchanging information between them.
(v) They send the results back from the accelerators to the cluster.

As with Kernel_call, subsequent calls to Kernel_call_multiple are possible. Moreover, both can be used in combination.

VI. Implementation

This section discusses the implementation of MPIX_Kernel_call and MPIX_Kernel_call_multiple. We implement both calls on top and not as part of the MPI library as this improves portability. Regarding the arguments of the two functions, the intracommunicator of the parents `comm` is not required for our implementation. However, it is given to enable the possibility for different implementations. As stated earlier, a function that is requested for execution must be available in the program that was previously spawned. Furthermore, our purpose is to free the programmer from additional work for enabling kernel execution on accelerator processes. Thus, we decided to let the application programmer only implement the minimum required, which is the kernel functions themselves. The rest of the logic is implemented by us in the main function of the spawned program. This program consists of two parts. One part contains the kernel functions provided by the programmer, while the other part contains the main function which is responsible for handling the kernel execution requests. Both parts are combined in the linking step which creates the actual binary program ready for spawning on the accelerators.

Note that cluster nodes and accelerators are not necessarily fully binary compatible. For instance, the DEEP system uses Xeon CPUs on the cluster nodes and Xeon Phi CPUs as accelerators. However, as the data types of the programming language and the numerical precision of the calculations are identical on both CPU architectures, the precision of final results is not affected. Otherwise, to account for different numerical precisions between cluster node and accelerator, algorithmic changes might become necessary. For this reason, it is only required to compile the binaries for the correct architecture, which is cluster node or accelerator. This is no limitation as the spawn call can be provided with a binary which is different from the main program.

Once the program was spawned on the accelerators, kernel requests can be sent, e.g., via MPIX_Kernel_call. Since this call is semantically identical to and thus implemented with MPIX_Kernel_call_multiple using a `count` argument of one, we focus on Kernel_call_multiple in the following. This function compiles a message from its arguments which is then sent from rank `root` of the cluster nodes to rank 0 of the spawned accelerator processes. The message contains, (i) the number of kernels to execute, (ii) the kernel names, (iii) the argument count for each kernel, (iv) the sizes of all the kernel arguments, and (v) the kernel arguments themselves. The message is transmitted via MPI_Send and received with MPI_Recv in the intercommunicator connecting cluster nodes and accelerators. A similar communication mechanism designed for implementing programming models is called active messages [11]. Here, instead of the function name, an active message carries the address of a request handler and the input data to be processed on message arrival. In our current implementation, we use a pre-allocated receive buffer of fixed size. This implies that the message cannot be larger than this size. However, a simple way to extend the protocol to allow for larger messages is to split the large message into smaller chunks which are then separately sent and received. Nevertheless, as this message carries usual kernel arguments only, it can be expected to be at most in the order of few megabytes. Larger amounts of data are recommended to be MPI-communicated from within the kernel itself.

Upon receipt of the complete message, the corresponding accelerator process broadcasts the size of this message to the remaining processes in its intracommunicator. This is necessary for them since they need to know the exact size of the message before they can actually receive it in a second broadcast operation. When the first broadcast starts, the single message from the cluster node to the accelerator process has already been delivered and Kernel_call_multiple returned. Consequently, the cluster node processes can asynchronously proceed with their execution and need not wait for the kernel to start. To account for the communication penalty between the cluster and the booster part of the DEEP system, we do not directly issue the broadcast but start with a single message between the process groups.

After the accelerator processes received the information on the requested kernels and their arguments, the next step is to call the first kernel. As there is no additional preprocessing of the user-implemented kernel functions involved in our approach, the kernel functions are not known before runtime to our protocol. At runtime, the only hint of their existence in the spawned program are the kernel names communicated through Kernel_call_multiple. For this reason, we use the kernel name to get a handle to the corresponding function to launch it. We accomplish this with the help of the dynamic loading programming interface, which is commonly used for implementing software plugins. The programming interface is part of the POSIX standard and thus available on many platforms. Here, we use the dlsym function, which takes the kernel name and returns the address of this kernel function in memory. Finally, we invoke the function using the address as a function pointer. As the number of arguments might differ among kernels and given that a usual function call must already contain all its arguments at compile time, it is not possible to call the kernel functions with their arguments with a classical function call. Instead, we push the kernel arguments manually onto the process stack before actually starting the kernel function. Note that the dlsym-based solution assumes that the kernel function name chosen by the application programmer in the program code remains the same for the compiled program. However, depending on the programming language, name mangling may change the symbol name of the function in the program binary. With C, we did not encounter any problems. To avoid this issue with Fortran kernel functions, we recommend using the BIND(C) attribute in their definitions. For C++ kernels, our approach is to declare them with extern "C". These workarounds are only required for the kernel function that forms the entry point to the actual kernel code. Additional lower-level functions within the kernel code are not affected.

After all accelerator processes have started the first kernel in the way described above, the corresponding function is executed and finally finished. After finishing the kernel, control is returned to our main function in the spawned program. If there are additional kernel requests, then the next kernel is started. Otherwise, the main function is waiting for the next incoming message containing new kernel requests. Once the spawned processes are not needed anymore, disconnecting from the parents and termination of the children is achieved via sending an empty string as kernel name for execution.

VII. Results

In this section, we investigate the startup overhead for launching parallel MPI kernels using the different approaches described above. In particular, we consider (i) the mulitple-spawns approach where every kernel call invokes one MPI_Comm_spawn, (ii) the single-spawn approach with one initial spawn followed by one MPIX_Kernel_call per kernel, and (iii) the single-spawn approach with one initial spawn followed by one MPIX_Kernel_call_multiple for a batch of kernels. Unfortunately, the complete DEEP system with its network-attached accelerators is not available yet. However, the cluster part of DEEP can already be used and serves as the testbed for our measurements. Details about the hardware of the cluster part have already been presented above. Note that using a usual cluster without the network-attached accelerators is not a limitation for our experiments, as we time the kernel startup mechanism which is not affected by the computational power of different types of nodes. The configuration for the timing measurements is the following. We use 120 cluster nodes in total. A subset of 40 nodes is used as the cluster nodes which want to offload parallel MPI kernels onto accelerators. The number of accelerators is 80, where the remaining cluster nodes take the role of these accelerators. This yields a cluster node to accelerator ratio of one to two. However, with network-attached accelerators we are not restricted to this ratio. We start two processes per cluster node, one per socket, which gives 80 cluster-node processes in total. These 80 cluster-node processes collectively start the same kernel on 80 accelerator processes. That is, each of the 80 accelerators runs one process. The software environment for our measurements is CentOS 6.4 (Linux 2.6.32) with the MPI library Open MPI 1.6.4 and NFS as network file system. The actual timing measurements were performed with the SKaMPI 5.0.4 MPI benchmark framework which uses a window-based methodology for more precisely timing collectives [12].

Figure 3 depicts the time it takes to start different numbers of kernels using the three different approaches. Note that while with MPIX_Kernel_call one message is sent per kernel execution, MPIX_Kernel_call_multiple sends the number of corresponding kernel requests in a single message. The arguments for each kernel consisted of five double precision values. To avoid overlap and thus wrong timings of successive collective operations, we measured the time for a single spawn and a single Kernel_call individually. The final times in the plot are then obtained by adding the individual times accordingly. The time for one spawn is the average of 30 individual measurements, while the number of measurements for Kernel_call and Kernel_call_multiple is 100. As can be seen in Figure 3, the Kernel_call and Kernel_call_multiple approaches clearly outperform the multiple-spawns approach. However, the timings of the two are very similar since they are dominated by the initial spawn which takes about 1.3 sec. Times similar to the initial spawn were obtained for optimized spawn implementations in [8]. The time for spawning consist of two major components: (i) loading the spawned program for execution and (ii) exchanging process information and connection setup between parent and child processes. That is, spawn performance depends on both the performance of the network filesystem for loading the spawned program and the interconnect for connection setup. However, because our solution loads the program only once while allowing multiple kernel calls, its performance depends primarily on the MPI message transfer latency and less on the network filesystem.

Note that the times obtained using the two kernel call functions are expected to be equivalent to the times based on a protocol implemented by the application programmer themselves. Figure 4 takes a closer look at the time differences of the individual Kernel_call and Kernel_call_multiple calls. As expected, it is shown that the benefit of Kernel_call_multiple increases with an increasing number of kernel calls. Note that the jump in time from 64 to 128 kernels might be due to the network hardware configuration. To get a more complete picture of the kernel startup overhead, we changed the configuration to 24 cluster nodes and 96 accelerators, where 48 cluster node processes start 96 accelerator processes. This yields a 1 to 4 ratio between cluster nodes and accelerators. However, the timings for this configuration were almost identical to the previous ones.

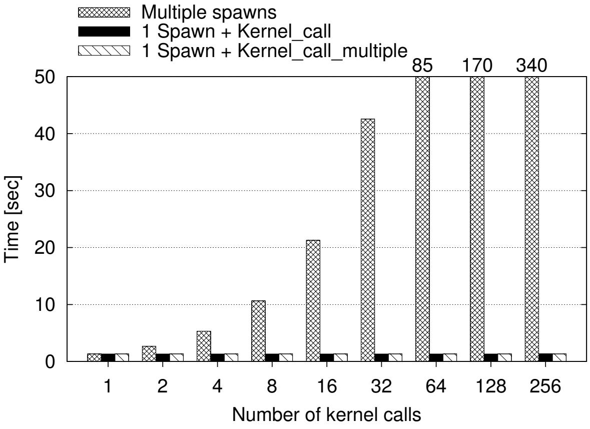

Figure 3. Kernel startup overhead for different numbers of kernel calls with different kernel offloading approaches.

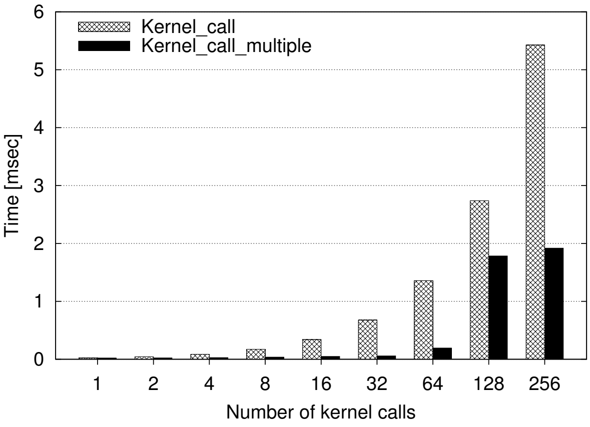

Figure 4. Detailed comparison of Kernel_call and Kernel_call_multiple without initial spawn.

VIII. Conclusion and Outlook

Network-attached accelerators, as they appear in the DEEP system, offer application developers the opportunity to better account for varying scaling characteristics within their codes. Even larger code regions that use MPI internally and bear the potential for massive node-level parallelism can be offloaded over the network. Within these kernels, MPI communication can be used for data exchange with other accelerators or with the cluster nodes that dispatched the kernel. However, current accelerator programming models such as CUDA or OpenCL do not support this offloading scenario. For implementing this scenario, we presented different offloading approaches based on MPI's MPI_Comm_spawn process creation mechanism. Starting with a pure spawn-based solution, we introduced the additional functions MPIX_Kernel_call and MPIX_Kernel_call_multiple which complement MPI's spawn call and compensate for its drawbacks. These two calls provide a convenient interface to the application programmer for starting MPI-based compute kernels on multiple accelerators. In addition, our experimental results show that they can noticeably reduce the kernel startup overhead compared to a pure spawn-based solution. Once the DEEP system is fully available, we will be able to actually run applications on this novel architecture using our proposed offloading mechanism. Currently, we assist application developers in their porting efforts. In the future, the insights gained from porting real codes might motivate facilities such as testing for kernel completion.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Program (FP7/2007-2013) under Grant Agreement n° 287530.

References

[1] D. Mallon, N. Eicker, M. Innocenti, G. Lapenta, T. Lippert, and E. Suarez, "On the scalability of the clusters-booster concept: A critical assessment of the DEEP architecture," in *Proc. of the Future HPC Systems*, 2012.

[2] S. Markidis, G. Lapenta, and Rizwan-Uddin, "Multi-scale simulations of plasma with iPIC3D," *Mathematics and Computers in Simulation*, vol. 80, no. 7, 2010.

[3] J. Duato, A. Peña, F. Silla, R. Mayo, and E. Quintana-Orti, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *Proc. of the Int. Conf. on High Performance Computing and Simulation (HPCS)*, 2010.

[4] S. Rinke, D. Becker, T. Lippert, S. Prabhakaran, L. Westphal, and F. Wolf, "A dynamic accelerator-cluster architecture," in *Proc. of the 41st Int. Conf. on Parallel Processing Workshops*, Sep. 2012.

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

[6] T. Kimura and H. Takemiya, "Local area metacomputing for multidisciplinary problems: A case study for fluid/structure coupled simulation," in *Proc. of the 12th Int. Conf. on Supercomputing*, 1998.

[7] M. Cera, J. Lima, N. Maillard, and P. Navaux, "Challenges and issues of supporting task parallelism in MPI," in *Proc. of the 17th European MPI Users' Group Meeting*, 2010.

[8] T. Gangadharappa, M. Koop, and D. Panda, "Designing and evaluating MPI-2 dynamic process management support for InfiniBand," in *Proc. of the 38th Int. Conf. on Parallel Processing Workshops*, Sep. 2009.

[9] S. Prabhakaran, M. Iqbal, S. Rinke, and F. Wolf, "A dynamic resource management system for network-attached accelerator clusters," in *Proc. of the 42nd International Conference on Parallel Processing Workshops (ICPPW 2013), Lyon, France*, Oct. 2013.

[10] H. Fröning, M. Nüssle, H. Leitz, C. Leber, and U. Brüning, "On Achieving High Message Rates," in *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2013.

[11] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proc. of the 19th Annual International Symposium on Computer Architecture*, 1992.

[12] T. Worsch, R. Reussner, and W. Augustin, "On benchmarking collective MPI operations," in *Proc. of the 9th European PVM/MPI Users' Group Meeting*, 2002.