Abstract—Exascale performance requires a level of energy efficiency only achievable with specialized hardware. Hence, to build a general purpose HPC system with exascale performance it will be necessary different types of processors, memory technologies and interconnection networks.

However most HPC applications use MPI to implement a rigid Single Program Multiple Data (SPMD) execution model that no longer fits the heterogeneous nature of the underlying hardware. MPI provides a powerful and flexible MPI_Comm_spawn API call that was designed to exploit dynamically heterogeneous hardware but at the expense of a higher complexity, hindering a wider adoption of this API.

In this work, we have extended the OmpSs programming model to dynamically offload MPI kernels, replacing the low-level and error prone MPI_Comm_spawn call with high-level and easy to use OmpSs pragmas. The evaluation shows that our proposal dramatically simplifies the dynamic offloading of MPI kernels while keeping the same performance and scalability as MPI_CommSpawn.

Keywords: HPC, MPI, collective offloading, OmpSs.

I. INTRODUCTION

Since the race to Exascale computing started and new architectures were proposed, supercomputers are evolving from homogeneous systems in which every node has the same hardware configuration to truly heterogeneous systems, where there are different sets of nodes for different purposes, and each node also contains different processors, memories and interconnection networks. A good example of this trend is the DEEP system [1] that is composed of a cluster of Xeon processors inter-connected with an Infiniband network and a cluster of Xeon Phi processors interconnected with an Extoll network. The Stampede supercomputer, hosted at the Texas Advanced Computing Center (TACC), is also composed of compute nodes with different characteristics: two Xeon processors and one or two Xeon Phi processors attached.

On these systems, the traditionally used Single Program Multiple Data (SPMD) execution model is not adequate to effectively exploit the underlying resources. Most applications have different computational phases, and each of these phases may run best on a different type and/or number of nodes.

Thus, many applications will benefit from a Multiple Program Multiple Data (MPMD) execution model, in which each computational phase run on top of the hardware that better suits its needs, effectively exploiting the heterogeneous nature of the underlying system. MPI [2] provides the MPI_Comm_spawn to properly implement an MPMD execution model. This API call enables the dynamic spawn of new MPI processes on additional compute nodes that can run a different program, which is connected and can communicate with the original one. MPI_Comm_spawn is very flexible and powerful, but also, complex and error-prone feature due to its low-level interface.

Additionally, the very nature of MPMD programs make them difficult to implement and reason about because the programmer must not only manually manages the intra-communications of each spawned MPI program, but also the intercommunications required between the different MPI programs.

As part of the DEEP project, we have extended the OmpSs [3] data-flow programming model to support the dynamic offload of MPI tasks, providing a practical way to implement MPMD applications without any of the complexities associated with the direct use of MPI_Comm_spawn. To that end, we have developed a simple API to dynamically allocate nodes/MPI processes, which returns a MPI intercommunicator that encloses all the newly created MPI processes. Additionally, OmpSs has been extended with a new onto(communicator, rank) clause used to offload the execution of tasks on the MPI process identified by the previous communicator and rank parameters.

II. OMPSS COLLECTIVE OFFLOAD

OmpSs is a directive-based programming model that enables the execution of sequential programs in a data-flow way. The programmer only needs to specify the data which is going to be read (in) and written (out) inside a function (task). Once this is provided, the code will be compiled by Mercurium compiler which will generate tasks to be executed by Nanos++ runtime.

OmpSs has a team of worker threads. The main thread will start executing the application and generating tasks which are added to a DAG (Directed acyclic graph), once a task has all its dependencies satisfied, it is moved to a ready queue. Then, each idle thread try to steal a ready task to execute it.
A. Dynamic Offloading with OmpSs

Allocation of remote nodes is quite expensive in terms of execution time. With our model, allocation is performed once for each group of MPI processes, then they can execute as many offloaded tasks as needed. Allocation can be easily done by using our API (which uses MPI_Comm_spawn multiple internally):

```c
void parallelCalcForces(particle_t* local, particle_t* tmp, force_t* forces, int np, int tsteps)
{
    int mpi_size;
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
    for (int t = 0; t < timesteps; t++) {
        particles_block_t* remote = local;
        for (int i = 0; i < mpi_size; i++) {
            calculate_forces(forces, local, remote, n_blocks);
            exchange_particles(remote, tmp, n_blocks, rank, rank_size, i);
            remote = tmp;
        }
        update_particles(n_blocks, local, forces, time_interval);
    }
}
```

In Listing 3 we can see a simple way to offload the calculate forces of this NBody benchmark to a dynamically allocated group of MPI processes, which can run on compute nodes with a different architecture. For the sake of simplicity, Listing 3 shows an N to O offload where each MPI process of the initial MPI_COMM_WORLD communicator offloads the calculation of the forces to another MPI process. However, our approach supports any N to M offload configuration.

### Experimental Evaluation

The objective of this evaluation Section is twofold. Firstly, to verify that the performance of an offloaded MPI kernel is not affected by the offloading processes, and secondly, to measure the performance of the dynamic allocation of MPI processes as well as the process of offloading MPI kernels.

#### A. Performance evaluation of offloaded MPI kernels

We executed the NBody benchmark shown in listing 2 on Stampede Supercomputer, to compare our approach with a traditional MPI code. In the host processor (Native Host), in the Xeon Phi (Native) and then we offloaded the algorithm using Xeon+Xeon Phi (OmpSs offload).

As seen in Figure 1 single-node offload performance is the same than native execution, so there are no performance
B. Performance evaluation of offloading mechanism

Once we have verified that the performance of MPI kernels is unaffected by the offloading, we focus on the performance evaluation of the offloading process. In order to do this, we measure the two possible sources of overhead, the one which DEEP_Booster_Alloc introduces compared with MPI_Comm_Spawn and the one in offloading MPI kernels and its parameters.

In Figure 2 we can see the time taken by each step of the spawn (1 master to X slaves), we see the overhead of initializing our runtime and offload structures is negligible compared with native MPI_Comm_spawn. In addition to this, scalability seems to be good when increasing the number of nodes, as the spawn can be done in parallel on each node.

In addition to allocation performance, we studied the performance of offloading simple tasks without computation nor big data transfers (with data cached in the remote node). In order to do this we launch 1000 simple tasks in 1, 2 and 4 remote processes running in a Xeon Phi, considering the time of a dummy MPI task as sending two messages (struct+integer) and receiving one integer. Native MPI achieves a peak throughput of 35,000 dummy tasks per second while OmpSs achieves a throughput of 25,000 dummy tasks per second.

We also implemented a simple benchmark which compares a more realistic worst-case scenario where all the data is sent and received from the node (which is hard to optimize compared with native) where all Native MPI tasks are hard-coded, something which in large programs where the control flow is more complex and depends on decisions taken in the host will not be easy to code. The amount of lines of code is 44 in OmpSs and 100 in native MPI.

At Figure 3 we can see that in our system the average overhead per task is negligible for large tasks, and quite small (around 3%) for small tasks. Ideally offloaded sections should be large tasks or kernels which can communicate between them using native MPI, so overheads should not be a critical problem.

IV. Conclusions

This extended abstract shows how OmpSs, a programming model that runs sequentially written applications, in a data-flow way, has been augmented with the capability of offloading MPI tasks to remote nodes dynamically spawned during the execution of an application.

With the use of a simple and concise syntax, OmpSs can offload these tasks to remote nodes. Our results show competitive performance when offloading MPI tasks, as the performance obtained is equivalent to the native execution of these MPI tasks and the performance loss in data and code transfer with a theoretical native MPI implementation is negligible. Thus, our offloading extension is very similar to the one provided by the Intel Offloading, but the last one is restricted to computational kernels that can not contain any call to MPI, while our collective offload fully support it.
REFERENCES

