OmpSs task offload

Florentino Sainz
Facultat d’Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya

A thesis submitted for the degree of
Master in Innovation and Research in Informatics (MIRI-HPC)

1st of July, 2014

Advisor: Vicenç Beltran Querol, BSC-CNS
Tutor: Jesús José Labarta Mancho, BSC-CNS & UPC-DAC
To my parents for their unwavering support, in all my endeavors.
Acknowledgements

I want to thank all people who helped me during this project, specially the ones involved in the development of OmpSs tools. I also want to thank Vicenç Beltran and Jesús Labarta for their guidance and support during this work.

The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n°287530 and 610476.
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 different types of processors, memory technologies and interconnection networks will be necessary. Heterogeneous hardware is already present on some top supercomputer systems that are composed of different compute nodes, which at the same time, contains different types of processors and memories. However, heterogeneous hardware is much harder to manage and exploit than homogeneous hardware, further increasing the complexity of the applications that run on HPC systems.

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. However, 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, which has hindered a wider adoption of this API.

In this master thesis, we have extended the OmpSs programming model to offload dynamically MPI kernels, replacing the low-level and error prone MPI_Comm_spawn call with the 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_Comm_spawn.
# Contents

1 Introduction ............................... 1

2 State of the art ............................. 3
  2.1 Intra-node heterogeneity .................. 3
  2.1.1 CUDA .................................. 4
  2.1.2 OpenCL .................................. 5
  2.1.3 Intel Offload ............................ 7
  2.1.4 Higher level programming models ........... 8
  2.2 Cluster level heterogeneity .................. 8
  2.2.1 rCUDA .................................. 8
  2.2.2 Virtual OpenCL .......................... 9
  2.2.3 MPI dynamic process spawn ................. 9

3 OmpSs .................................... 11
  3.1 Tasks .................................... 11
  3.2 Mercurium compiler ........................ 13
  3.3 Nanox Runtime .............................. 15

4 Implementation of the OmpSs Offload ............ 16
  4.1 Nanox Runtime .............................. 18
  4.1.1 Offload mechanism ....................... 18
  4.1.2 Data management .......................... 19
  4.1.3 Threads and scheduling .................... 19
  4.1.4 Dynamic compilation of offload Plugin ....... 20
  4.2 Mercurium compiler .......................... 21
4.2.1 Clause extensions .................................. 21
4.2.2 Task generation .................................. 21
4.2.3 Offloading global variables ....................... 23
4.2.4 Add custom compilers ............................ 23
4.2.5 Dynamic compilation of offload Plugin .......... 23
4.3 Executable generation .................................. 24

5 Evaluation ............................................. 26
5.1 Performance of offloaded code ....................... 26
5.2 Overhead comparison with native MPI ............. 33
5.3 Real applications .................................... 39
  5.3.1 SRMIP ......................................... 40
  5.3.2 TurboRVB ...................................... 43
  5.3.3 FWI ............................................. 44

6 Conclusions and Future Work ......................... 47
List of Figures

3.1 Mercurium compilation flow .................................................. 14

5.1 Calculate forces particle communications among iterations. Represents how each rank owns a local partition of the particles which will be exchanged with other nodes in order to calculate the accumulated forces for that partition .................................................. 28

5.2 OmpSs NBody Simulation offload (Single iteration communications). Master processes will spawn two worker processes which can communicate internally in the Xeon Phi and have them calculating the forces. Total Communications = 2N/NP + N ........................................ 29

5.3 Intel Offload NBody Simulation (Single iteration communications). Master processes will spawn one worker process each which can’t communicate internally in the Xeon Phi and have them calculating the forces. Total Communications = N/NP + 2N ........................................ 31

5.4 Single node performance of each of the different versions of the NBody Benchmark. We can see how it’s slower in the host and all the other versions, including our offload, obtain similar performance. ............ 32

5.5 Weak Scaling Speedup of different versions of NBody Benchmark . 32

5.6 Strong scaling speedup of different versions of NBody Benchmark . 33

5.7 OmpSs vs Native MPI Spawn. Each bar shows how time is distributed in MPI Spawn time, which is the time that MPI takes to spawn the processes, and the extra operations we introduce to setup our offload mechanism. .......................................................... 34

5.8 OmpSs vs Native MPI execution .................................................. 39
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9</td>
<td>SRMIP master-to-workers partitioning and work sharing before applying Offload</td>
<td>40</td>
</tr>
<tr>
<td>5.10</td>
<td>SRMIP master-to-workers partitioning and work sharing after applying Offload</td>
<td>41</td>
</tr>
<tr>
<td>5.11</td>
<td>FWI execution 1. Shows how work is offloaded to level 2 nodes and how level 3 nodes calculate the shots</td>
<td>44</td>
</tr>
<tr>
<td>5.12</td>
<td>FWI execution 2. Shows how each group of 4 level 3 nodes process different shots at the same time independently</td>
<td>46</td>
</tr>
</tbody>
</table>
List of Tables

5.1  NBody Execution Device Schema ..................................... 29
5.2  Comparison of the number of code lines with and without OmpSs offload 43
Chapter 1

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 Stampede supercomputer hosted at the Texas Advanced Computing Center (TACC). This supercomputer is mainly composed of compute nodes with two Xeon processors and one Xeon Phi processor attached, but there are also compute nodes with two Xeon Phi processors instead of one, compute nodes with 1TB of memory or compute nodes equipped with NVIDIA K20 GPUs. Another example is the DEEP system [1] that is composed of a cluster of Xeon processors linked with an Infiniband network and a cluster of Xeon Phi processors linked with an Extoll network.

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. For instance, one compute phase may scale poorly with the number of nodes, so it better runs on a small number of powerful nodes, while other phases may scale very well and are more efficiently executed on a large number of nodes with accelerators.

Thus, to effectively exploit an heterogeneous system, most applications should have a
1. INTRODUCTION

Multiple Program Multiple Data (MPMD) execution model, in which each computational phase runs on top of the hardware that better suits its needs. MPI provides the `MPI_Comm_spawn` to properly implement an MPMD 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. However, the usage of this low-level API is complex and error prone.

The very nature of MPMD programs make them difficult to implement and the reason about because the programmer must not only manually manage the intracommunications of each spawned MPI program, but also the intercommunications required between the different MPI programs. These include the data transfers across MPI programs as well as the necessary synchronizations to orchestrate the whole program execution. The complexity of implementing this approach is reflected by the low number of HPC applications that currently has an MPMD execution model.

We have extended our OmpSs [2] 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` API. 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(comm, rank)` clause that specify in which specific MPI process a task has to run. Our extended compiler and runtime system transparently manage all data transfers and synchronizations required.
Chapter 2

State of the art

The growing popularity of hardware accelerators has encouraged academic and industrial researchers to develop novel programming models to make the most of these new highly parallel compute devices with a moderate effort. The applications that run on these hybrid systems, composed of multi-cores and hardware accelerators, usually have some parts that do not scale well and run better on multi-cores while other highly parallel parts can exploit the full potential of hardware accelerators. Hence most parallel programming models explicitly provide mechanisms to split and coordinate applications to successfully run on hybrid systems. The rest of the section divides previous research efforts on this topic in intra-node and inter-node heterogeneity.

2.1 Intra-node heterogeneity

At the intra-node level, much work has been done and several programming models have been implemented, like OpenCL [3], CUDA [4], Intel Offload, OpenACC or OpenMP 4.0. All of them provide mechanisms to divide applications in two parts, the one that runs on the multi-core side, which is usually known as the host part, and the part that runs on the accelerator, that is usually known as a kernel.

CUDA was the first mainstream programming model widely used to exploit GPUs. A standardization effort to exploit hardware accelerators have led to the development of OpenCL, which is the only programming model that can exploit a variety of accelerators, including the Intel Xeon Phi accelerators. In the rest of this Section, we will summarize the most widely used programming models to exploit local accelerators.
2. STATE OF THE ART

2.1 Intra-node heterogeneity

2.1.1 CUDA

CUDA (formerly Compute Unified Device Architecture) was first introduced back in 2007 by NVIDIA to exploit the GPUs computational power.

CUDA is composed by a group of development tools created by NVIDIA which allows programmers to use a programming language similar to C and C++ in order program GPUs.

The language provides a host API which is used to copy data to/from the GPU and launch kernels inside the GPU, and a C-like programming language, in which kernels are written. These kernels are pieces of code/functions which will run inside the GPU.

Listing 2.1: Kernel written in CUDA C

```c
__global__ void saxpy(int n, float a, float* x, float* y)
{
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    if(i < n)
        y[i] = a * x[i] + y[i];
}
```

Listing 2.2: CUDA program using host API to launch kernels

```c
int main(void)
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));

    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));

    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY on 1M elements
```
2. STATE OF THE ART

2.1 Intra-node heterogeneity

```c
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

float maxError = 0.0f;
for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
```

At listing 2.1 we can see how the programmer will obtain the id of the thread (consecutive numbers) and will make each thread handle a different element of the vectors. The amount of threads launched in the GPU will be specified by using the host API as seen at listing 2.2 at the time of launching the kernel.

2.1.2 OpenCL

OpenCL was first developed by Apple and then submitted to the Khronos group, which published the first specification at the end of 2008. Its philosophy and features are similar to CUDA but it also covers other devices (CPU, Xeon Phi, FPGAs, GPUs...). The complexity of developing an OpenCL application is, like in CUDA, clearly split in two different parts.

The first one is related to the development of optimized OpenCL C kernels like the `saxpy` kernel we can see in Listing 2.3 that can exploit accelerator hardware. The second one includes compiling the kernel, calling it and moving data from the host to the accelerator and viceversa, for this, OpenCL provides a low-level API to interact with the accelerator.

```
Listing 2.3: Kernel written in OpenCL C
__kernel void saxpy(int n, float a,
    __global float* x, __global float* y)
{
    int i = get_global_id(0);
    if(i < n)
        y[i] = a * x[i] + y[i];
}
```

```
Listing 2.4: OpenCL program using host API to launch kernels
```
```c
int main(int argc, char** argv)
{
    float a, h_x[1024], h_y[1024];
    // Init a, h_x and h_y;
    cl_uint numPlats;
    clGetPlatformIDs(0, 0, &numPlats);
    cl_platform_id Plat[numPlats];
    clGetPlatformIDs(numPlats, Plat, 0);
    clGetDeviceIDs(Plat[1], DEV, 1, &id, 0);
    cl_context ctx =
        clCreateContext(0, 1, &id, 0, 0, 0);
    cl_command_queue cmd =
        clCreateCommandQueue(ctx, id, 0, 0);
    cl_program program =
        clCreateProgramWithSource(ctx, 1, KernelSrc, 0, 0);
    clBuildProgram(program, 0, 0, 0, 0, 0);
    cl_kernel ko_saxpy = clCreateKernel( program, "saxpy", 0);

    cl_mem d_x = clCreateBuffer(ctx, 0, sizeof(float) * n, 0, 0);
    cl_mem d_y = clCreateBuffer(ctx, 0, sizeof(float) * n, 0, 0);

    clEnqueueWriteBuffer(cmd, d_x, CL_TRUE, 0, sizeof(float) * n, h_x, 0, 0, 0);
    clEnqueueWriteBuffer(cmd, d_y, CL_TRUE, 0, sizeof(float) * n, h_y, 0, 0, 0);

    clSetKernelArg(ko_saxpy, 0, 4, &n);
    clSetKernelArg(ko_saxpy, 1, 4, &a);
    size_t sze = sizeof(cl_mem);
    clSetKernelArg(ko_saxpy, 2, sze, &d_x);
    clSetKernelArg(ko_saxpy, 3, sze, &d_y);

    size_t global = 1024, local = 128;
    clEnqueueNDRangeKernel(cmd, ko_saxpy,
        1, 0, &global, &local, 0, 0, 0);

    clFinish(commands);

    clEnqueueReadBuffer( commands, d_y, CL_TRUE, 0 * n, h_y, 0, 0, 0 );
}
```
2. STATE OF THE ART

2.1 Intra-node heterogeneity

```c
clReleaseMemObject(d_x);
clReleaseMemObject(d_y);
clReleaseProgram(program);
clReleaseKernel(ko_saxpy);
clReleaseCommandQueue(commands);
clReleaseContext(ctx);
return 0;
```

At Listing 2.3 we can see how the programmer will obtain the id of the thread (consecutive numbers) and will make each thread handle a different element of the vectors. The amount of threads launched in the GPU will be specified by using the host API as seen at the time of launching the kernel as seen at listing 2.4. We can also see that the verbosity of this host API is quite high.

2.1.3 Intel Offload

Intel Offload is a technology developed by Intel which allows users to run C/C++ code at Intel Xeon Phi accelerators which are in the same node as the host Xeon E5.

It provides an extension to the compiler which is able to interpret pragmas which specify the regions of code which will be executed on remote devices. Once the execution reach these regions of code, it will stop the program and transfer all the data needed to the Xeon Phi and begin the execution of that code. Once the code is finished, data will be copied back to the host and the execution will continue.

Listing 2.5: Intel Offload code

```c
void saxpy(int n, float a, float* x, float* y)
{
    #pragma offload target(mic:0) in(x : length(n) alloc_if(1)
                     free_if(1)) inout(y : length(n) alloc_if(1) free_if(1))
    for (int i=0; i<n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

At Listing 2.3 we can see how the programmer will execute the function specifying that the code will be offloaded to a "mic" (Xeon Phi) and the data it needs. After the

---

7
offload code has finished the host CPU will finish the function and continue with the regular code.

**2.1.4 Higher level programming models**

The complexity of programming in CUDA or OpenCL can be alleviated by using some higher level programming models, like OpenMP 4.0 [5], OpenACC [6] which substitute APIs for pragmas and transform sequential C code to CUDA/OpenCL kernels. Other languages like OmpSs do not generate kernels, but they handle the integration of the kernel with the main application in a transparent way.

**2.2 Cluster level heterogeneity**

The previous programming models were not designed to work on distributed environments. However, there are a few technologies such as rCUDA [7] or Virtual OpenCL [8], which extends CUDA and OpenCL respectively and allow programmers to transparently use remote devices as local devices. Additionally, MPI-2 standard already included some functionalities to deal with heterogeneous cluster, but not with hardware accelerators in mind.

**2.2.1 rCUDA**

rCUDA is a middle-ware that enables CUDA remoting over a commodity network. That is, the middle-ware allows an application to use a CUDA-compatible graphics processing unit (GPU) installed in a remote computer as if it were installed in the computer where the application is being executed. rCUDA intercepts calls to the CUDA API in an application and sends them to a client(application)-server(node with GPUs) distributed architecture which allows them to execute CUDA code in remote nodes without having a local GPU. This technology can help to save energy or have clusters with very heterogeneous architectures, but still has the same problem than native CUDA, devices cannot communicate inside of a kernel and communicating between two different devices has to be done using a host CPU.
2. STATE OF THE ART

2.2 Cluster level heterogeneity

2.2.2 Virtual OpenCL

The VCL cluster platform is the OpenCL equivalent to rCUDA. However, in both technologies the transparent virtualization of the hardware accelerators may prevent programmers to make the most of them at some scenarios, as one usual bottleneck is the data-transfer between the host and the accelerator.

2.2.3 MPI dynamic process spawn

MPI-2 standard introduced a call (MPI_Comm_spawn) to enable the dynamic creation of new MPI processes. This API call enables the creation of dynamic and malleable distributed applications that can even run on heterogeneous clusters. This flexibility comes at the expenses of a large increase on the application complexity because the programmer must explicitly coordinate both MPI parts, which do not follow the usual SPMD execution model of most MPI applications. The lack of heterogeneous clusters until recently and the high complexity to effectively use this features has hindered a wider adoption of this technique.

Nowadays, programming models only support homogeneous clusters of hardware accelerators, where MPI is combined with a programming model that supports hardware accelerators at the node level. However, the emerging number of heterogeneous clusters, which combine different types of nodes with hardware accelerators, requires new approaches to effectively exploit them. In order to exploit cluster-level heterogeneity with many nodes with different characteristics, we propose our OmpSs Collective Offload model in which one or many nodes can offload work to a group of remote nodes which will execute a different part of the algorithm (a MPI kernel), while being able to directly communicate between them. Thus, our approach is as simple as the Intel Offload but as powerful as the MPI_CommSpawn.

One example of this kind of architectures is the one proposed under DEEP project in which a stand-alone cluster of Intel Xeon Phi accelerators connected with a high performance network will be used to offload MPI Kernels from another cluster of Xeon processors.

With the new Intel Xeon Phi based accelerators, there is a new scenario, an accelerator which can execute x86 code and supports MPI. In this scenario, programmers are able to run their applications in this accelerator with minor changes. In order to
get their best performance, parallel code should run on the accelerator and serial code on regular CPUs. Both Intel OpenCL or Intel Offload Model [9] were implemented in order to address this issue, but they are limited in the sense that accelerators cannot communicate between themselves. This limitation can be overcome by offloading using the mechanisms offered by MPI dynamic process spawn.
Chapter 3

OmpSs

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 Nanox++ runtime.

3.1 Tasks

Data dependencies are specified with the clauses in/out/inout which specify how data is accessed in a task and can be used to build a graph which controls that there are no race conditions between tasks, thus warranting a correct execution. The way dependencies work is that following instantiation order, a new task will not be able to start if a previous task is writing into the same data (out/inout) or if the task has to write into data (out/inout) which is being used by some previous task which is still executing.

We can see an example of how OmpSs works in Listing 3.1. 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 these tasks have all the dependencies satisfied, they are moved to a ready queue. Threads will steal these tasks from the ready queue and execute them. In our code, we can see that the first task outputs forces and the second task has forces as input, this means that second task will not start until first task finishes. By looking at dependencies we can see that almost
void parallelCalcForces(particle_t* local, particle_t* remote, 
force_t* forces, int np, int tsteps) {
int rank_size;
MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

for (int t = 0; t < timesteps; t++) {
    particles_block_t * remote = local;
    for(int i=0; i < rank_size; i++){
        #pragma omp task in([n_blocks] local, [n_blocks] remote)
            inout([n_blocks] forces)
        calculate_forces(forces, local, remote, n_blocks);

        #pragma omp task in([n_blocks] remote) out([n_blocks] tmp)
            exchange_particles(remote, tmp, n_blocks, rank, rank_size, i)
            remote=tmp;
    }
    #pragma omp task inout([n_blocks] local ) inout([n_blocks] forces)
        update_particles(n_blocks, local, forces, time_interval);}

#pragma omp taskwait}
every task is dependant from the previous one, but communication and computation can be overlapped. In OmpSs tasks can be nested, so calculate_forces function can be a highly parallel function too.

The program will wait for every task to finish by using pragma omp taskwait directive.

3.2 Mercurium compiler

Mercurium is a source-to-source compilation infrastructure aimed at fast prototyping. Current supported languages are C, C++ and Fortran. Mercurium is mainly used in Nanox environment to implement OpenMP but since it is quite extensible it has been used to implement other programming models or compiler transformations, examples include Cell Superscalar, Software Transactional Memory, Distributed Shared Memory or the ACOTES project, just to name a few.

Extending Mercurium is achieved using a plugin architecture, where plugins represent several phases of the compiler. These plugins are written in C++ and dynamically loaded by the compiler according to the chosen configuration. Code transformations are implemented in terms of source code.

Internally, the compiler is divided in the front-end, which uses bison to parse the source code and transform it into a tree which will be analysed or modified by the back-end. And the back-end, in which plug-ins and phases are implemented. During this project two phases were implemented:

- A simple phase which adds a function call to the first line of the main/PROGRAM and will be used by our “workers” and also by some other extensions of OmpSs whenever they have to change the behaviour of the user’s main.

- The main device phase which handles the code generation of offload tasks.

At Figure 3.1 we can see how mercurium will compile a program, the input source will be parsed by the frontend and the tree will be generated, after doing so, this code will be processed by the OmpSs/OMP code generator, which will process pragmas/tasks in case the code has it and will generate some parts of code needed by the runtime
Figure 3.1: Mercurium compilation flow
in order to execute a task. After doing so, the device provider (specified with the target or onto clauses) will generate the code needed by that particular device in order to execute the task and call any special compiler if needed by that concrete device.

After all the code has been generated (by modifying the tree), Mercurium will write this tree into a file or multiple files which will be compiled by the native compiler or with compilers specified by the devices, after doing so, Mercurium will merge all the object files into a single object file/executable.

3.3 Nanox Runtime

Nanox is a runtime designed to support parallel environments, mainly OmpSs.

Nanox provides services to support task level parallelism using data dependencies as a way to synchronize them. These tasks are implemented as execution threads whenever possible. In addition, Nanox supports keeping coherency among different memory spaces (like GPUs, remote nodes...).

The main objective of Nanox is to be used to develop new parallel environments. In order to help with that task, its extensible by plugins which can implement different features:

- Task scheduling
- Support to other devices
- Instrumentation
- Resource management

Many of the features of Nanox are also implemented as plugins. Like support to CUDA/OpenCL devices, Extrae tracing and others.

However, Nanox is not designed to be used directly by application developers, but instead as a backend for Programming Models such as OmpSs and OpenMP. These programming models are supported by Mercurium Compiler.
Chapter 4

Implementation of the OmpSs Offload

In this chapter we will cover modifications done to our runtime and compiler in order to perform OmpSs offloading.

We already saw an example of how OmpSs works in Listing 3.1. Our objective with dynamic offloading is that regular OmpSs tasks which are normally executed in the local node can be offloaded to remote MPI processes transparently.

We have augmented OmpSs with a new API call to dynamically allocate nodes (deep_booster_alloc) and a new OmpSs clause (onto) to easily offload task to these newly allocated nodes. In Listing 4.1 we can see how both the API and the clause are used in order to perform a vector sum in an allocated remote node. A detailed description of the new API call and OmpSs clause are provided in next Section. In order to execute this vector sum we have to perform the following actions, which will be performed by OmpSs toolchain:

- Mercurium will generate one binary for each target architecture (in this case one binary for Xeon and one for Xeon Phi). The binaries can be started either in master or slave mode. There is only one binary that starts in Master mode, which then executes the main/PROGRAM of the application. Once the main application allocates additional MPI processes the corresponding binary in SLAVE mode is executed. The binaries executed in slave mode, after initialization, waits for orders from the master.

- After compiling, during the execution of the master, user will perform the allocation of the slaves using deep_booster_alloc API call (which uses MPI_Comm_spawn_multiple).
4. IMPLEMENTATION OF THE OMPSS OFFLOAD

Listing 4.1: Offload vector sum

```c
void main()
    
    MPI_Comm workers;
    deep_booster_alloc(MPI_COMM_SELF, 1, 1, &workers);
    int a[N], b[N];
    initVectors(a, b);
    #pragma omp task in(b[0:N]) inout(a[0:N]) onto(workers,0)
    {
        #pragma omp parallel for
        for (int i=0; i<N;++i) {
            a[i]=a[i]+b[i];
        }
    }
    #pragma omp taskwait
    printVector(a, N);
    deep_booster_free(&workers);
    MPI_Finalize();
```

- Once the remote node is available and all dependencies are satisfied, local Nanox++ will send both arrays to the remote node and will send the order to execute the task, which after every data has been received, can be executed.

- After the task finishes, the remote node will send a signal to the master, which will free the dependencies blocked by that task.

- After the dependencies are freed, the taskwait will finish and Nanox++ will copy the data back to the host. After doing so, the main program can continue printing the results and freeing the nodes.

This is a simple example with no communications between the nodes, but in real programs, communications may be performed between offloaded tasks. More elaborated examples are shown later in Evaluation Section.
4. IMPLEMENTATION OF THE OMPSS OFFLOAD

4.1 Nanox Runtime

Nanox++ runtime had to be extended to allow allocation of remote nodes and task offloading to these nodes. In order to achieve this some modifications had to be done:

4.1.1 Offload mechanism

In order to allocate remote nodes, \texttt{MPI\_Comm\_spawn\_multiple} MPI call is used, this call is not user-friendly and it is not widely used, but it can be very useful for heterogeneous architectures. In Listing 4.1 it’s used when the user calls our APIs to allocate nodes.

Allocation is quite expensive in terms of execution time, with our model, allocation is performed once for each node/group of nodes, then they can be reused as many times as needed.

In order to allow users to easily allocate nodes, we extended Nanox++ so it provides a new user-level API which allows to allocate remote nodes, this API has the following interface:

Listing 4.2: Offload node reservation API

\begin{verbatim}
DEEP_Booster_alloc(MPI_Comm SpawningIntercomm, int NNODES, int PPN, MPI_Comm SpawnedIntercomm);
DEEP_Booster_free(MPI_Comm SpawnedIntercomm);
\end{verbatim}

\textit{DEEP\_Booster\_alloc}: This call allows users to allocate a group of remote nodes for task execution. The environment variables of the remote nodes, as well as their physical location can be specified by using a MPI-like hostfile or an environment variable. Apart from allocating nodes using \texttt{MPI\_Comm\_spawn\_multiple}, it creates every Nanox++ structure in both parts of the offload (threads, cache, etc) needed in order to manage the remote node internally. More than one parent node may allocate and launch tasks on a single remote node, but only one of them will be able to launch tasks on it at the same time. \texttt{NNODES} specifies the amount of nodes which the user wants to allocate and \texttt{PPN} specifies the number of processes per node which will be allocated.
4. IMPLEMENTATION OF THE OMPSS OFFLOAD

4.1 Nanox Runtime

Deep Booster free: This call allows users to free remote nodes which were allocated before, from this point they can not be used to offload anymore.

4.1.2 Data management

When executing tasks on remote nodes, data has to be moved to the remote location so it is available when executing these tasks, in order to perform this movement, Nanox++ already has a directory/cache which manages data from different memory spaces. In Listing 4.1 it is used when the data has to be moved from the master node to the remote node prior to the execution of the task and when the data comes back to the master node in the taskwait.

In order to benefit from Nanox++ cache, our device has to implement most basic operations (allocate, free and copies), both in the local node and in the remote node, so we implement every function needed in order to dispatch copies from the local node, these functions will send an MPI message to a daemon which will be running in the remote node, this daemon will do the remote allocation/free, or in case of data transfers, place them in the right place.

In data transfers, there is an important optimization, which is data shared between different tasks which will be executed on different remote nodes. In this case, instead of sending the data through the host node, we will send one instruction to each node ordering them to transfer the data directly using the network between both nodes, which should be faster than the one which connects the remote node with the host.

In addition to managing single-level cache, Nanox++ runtime will run in both sides of the offload (local and remote), so when passing data from one level to another, caches of both runtimes have to be consistent, this is useful whenever the user wants to do multi-level offloading or exploit other separate memory address space devices, like GPUs, inside the remote node.

4.1.3 Threads and scheduling

Nanox++ can balance load between different remote nodes dynamically, so if the programmer does not specify in which node each task must be ran, they will be launched
on any node which is available. In order to be able to do this, a thread representing these remote nodes has to be implemented inside the runtime. This thread will make sure that all the data is on the remote device and dependencies are satisfied before sending the task. In Listing 4.1 it is used to manage dependencies and how and when the task is executed, also to manage when the task finishes and the taskwait can proceed.

Scheduling policies also had to be modified, so programmer can specify in which node he wants to enforce the task to be running on. This can be done by using the onto clause which can be seen in Listing 4.1. In this clause, there are three possible scenarios:

- No restrictions: Nanox will choose any of the offloaded nodes to execute and are currently free.
- Only communicator: Nanox will choose any free node to execute the task, restricted to the the ones which are in this communicator and are not executing anything.
- Communicator+Rank: Nanox will only run this task on the specified node.

### 4.1.4 Dynamic compilation of offload Plugin

One of the main problems that appear when developing libraries which use MPI is that they have to work with any of the implementations of the MPI Standard. But each implementation has a different header and use different kind of structures to represent the data used by MPI (MPI_Comm,...), so they are not binary compatible with each other.

A workaround for this problem would be to compile one version of OmpSs for each implementation of MPI. But there are many implementations and versions, specially in HPC world, where each provider usually gives a optimized version for their machine, so this solution would not be easy to maintain.

Our solution to this problem is to make the device as a Nanox Plug-in which is totally independent from the core and is not compiled together with it. Whenever Nanox is installed, the sources of the offload plug-in will be distributed instead of
being compiled. In addition to the sources, the flags and the compiler name which were used to compile Nanox, will be saved into the installation folder.

These sources and flags will be later used by Mercurium to compile the plugin with the same flags and compiler than Nanox, and also with the same version of MPI that the user uses to compile his application.

### 4.2 Mercurium compiler

Currently, OmpSs programming model is already supported by Mercurium, so most of the code transformations needed to generate the source code of a task is provided by the base version of the compiler, this includes copying and allocating private variables, placing them in a data structure, and generating the information which Nanox++ needs to manage data and dependencies. Mercurium also detects when scalar variables have to be moved (firstprivate) from the host to the device, like $N$ in the previous sample.

#### 4.2.1 Clause extensions

Mercurium has to support the *onto* clause as seen in Listing 4.1, so support for this extra clause was added in the compiler. If this onto clause is detected, the *mpi* offload device is assumed, so it will be processed by our phase.

#### 4.2.2 Task generation

When generating tasks, an outline function/procedure is generated. The behaviour is different depending on the task:

- **Outline tasks:** this outline function is a wrapper which translates some pointers when needed by the data management mechanism and calls the original user function/procedure.

- **Inline tasks:** the task region is encapsulated into a function/procedure, variables are renamed so they match the original user code, and after doing so, the outline function can call this encapsulated function as if it was an outline task.
4. IMPLEMENTATION OF THE OMPSS OFFLOAD  4.2 Mercurium compiler

In our MPI offload approach, this outline function is split in two different parts, a host function which offloads the tasks and waits for it to finish, and a remote/offloaded function which will actually execute the code.

When passing a task from the host to the offload, arguments have to be sent, in order to do this, we pack every immediate/pointer argument into a C/Fortran structure/type and create an MPI datatype which sends them to the offloaded function. This allows us to send these arguments in a single message while allowing MPI to translate datatypes whenever it is needed (different architectures).

The generated outline function for offload tasks follows this flow:

1. Call host code of a task
2. Send task identifier to the remote node
3. Pack task arguments and send it to the remote node via MPI
4. Ask Nanox++ to translate array/copies addresses from local pointers to their equivalent remote pointers
5. Execute remote code of the task
   - Receive data structure
   - Execute outline with data structure
   - Send task finish signal to the host
6. Receive task-end signal

When tasks are generated, our compiler will store both the host function and the remote function into an array which is merged at linking time, so when generating the executable files, every task which was generated in our executable or any library pointed by it will be available on this unordered array.
4. IMPLEMENTATION OF THE OMPSS OFFLOAD  4.2 Mercurium compiler

4.2.3 Offloading global variables

During previous steps, we did not mention global variables, which are not initialized in the remote nodes, as we explained before, in order to offload these variables, they are privatized and their value gets packed inside the offloaded region. This works fine for most cases, but if the code of the task has a call to other procedure/function and this other procedure/function uses these global variables the value of those variables will not be available on that procedure/function.

To fix this problem, before calling a task, we set the global variable pointing to the local variable used inside the task, this way we make sure that every function will be accessing the same variable and the right value will be copied back to the host whenever it is needed. This is safe because only one task can be offloaded at a time on the same node, and using OmpSs programming model, if you are writing to the same global variable in two tasks, they will be serialized, so they will not execute concurrently on different nodes.

4.2.4 Add custom compilers

In addition to these changes, three new profiles have been added to Mercurium. mpimcc (C), mpimcxx (C++) and mpimfc (Fortran). These profiles allow to automatically link with MPI (needed in order to perform offload) and are capable of generating code for both architectures (Intel k1om and x86). Using these compilers, generating the executables should be as easy as running the same compilation sequence with and without using the flag –mmic.

4.2.5 Dynamic compilation of offload Plugin

As explained in Section 4.2.1, Nanox does not compile with Offload support but distributes the sources and the compilation flags.

Mercurium offload profile will compile user code normally by using native MPI wrappers, as any other profile would do.
4. IMPLEMENTATION OF THE OMPSS OFFLOAD

4.3 Executable generation

Whenever an executable has to be generated, Mercurium will read the flags distributed by Nanox installation and compile the Offload Plugin, which will be linked with user code and Nanox.

This way, the Offload Plugin (which is the part of Nanox which depends on MPI) will be compiled with the MPI implementation available in the environment, allowing users to use any MPI available in the system instead of being constrained to the one which was used to compile Nanox Runtime.

4.3 Executable generation

In order to offload, the programmer has to provide as many executables as architectures present in the offload, all of them named with the convention binary.arch. As previously stated, we provide the compiler executables to do this easily with k1om/mic and x86_64/intel64 architectures, but this implementation may be used with other architectures as long as this name convention is respected.

This executables are generated by our compiler, and they have two different behaviours, the slave behaviour and the one which executes the original application code.

In the slave behaviour, which is enabled automatically by our offload system, the executable will act as a daemon process which receives orders from another process and executes whatever tasks are needed. As we want to keep non data transfers overheads as low as possible, the message which initializes tasks will be an integer. This integer is the index of the task in our array of tasks. So finding which task we have to execute is as fast as finding the index of the task in the host array, sending it through the network and accessing the remote index the device in array.

As stated previously, in the remote node/accelerator, our runtime is available, which means that our programming model is also available inside the remote node. This allows the programmer to take advantage of OmpSs by using tasks also in the remote node, including any CUDA or OpenCL device which is available in that node.

This technique for indexing tasks, even though it is fast, has a problem, as we previously stated, these arrays have no particular order, so it has to be synchronized at initialisation time. When nodes are allocated, after the remote process is initialised,
4. IMPLEMENTATION OF THE OMPSS OFFLOAD

4.3 Executable generation

some data structures are sent from the host node to the remote node. By using this data structures, device will make sure that its tasks are on the same order than on the host, so our way to identify task by using an ID works correctly.

With this behaviour, we get a small overhead at node allocation time, which only happens a few times (usually one or two times at initialization time), but improve the performance on each task call, which are executed many times.
Chapter 5

Evaluation

The objective of this Chapter is to compare the performance of offloaded MPI kernels and with native MPI offloading. In addition we will show some real applications which were ported to our Offload model.

5.1 Performance of offloaded code

Our main objective is to show that the performance of our approach is similar to executing the algorithm natively. We have evaluated our approach with a N-body simulation benchmark implemented using different programming models or code distribution between host and offloaded parts.

These results have been taken on Stampede Supercomputer using Intel compilers and Intel MPI library. Our (per-node) test configuration consists in:

<table>
<thead>
<tr>
<th>Main Processor</th>
<th>2x8 Xeon E5-2680 2.7GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Per Host</td>
<td>32GB 8x4G 4C 1600MHz</td>
</tr>
<tr>
<td>Coprocessor</td>
<td>1 Xeon Phi SE10P 1.1GHz</td>
</tr>
<tr>
<td>Co-processor Memory</td>
<td>8GB GDDR5</td>
</tr>
<tr>
<td>Host-Co interconnect</td>
<td>QPI 8.0 GT/s PCI-e</td>
</tr>
</tbody>
</table>

Network protocol used for the tests was Ethernet emulated over infiniband because no direct MIC-to-MIC physical communication over infiniband is available yet.

As seen in Listing 5.1 the computational part of our implementation is divided in two different parts, calculate forces $O(n^2)$ which calculates the forces produced between the particles for one iteration and update particles $O(n)$, which based on previous
5. EVALUATION

5.1 Performance of offloaded code

Listing 5.1: NBody OmpSs offloaded

```c
void parallelCalcForces(particle_t* local, particle_t* tmp, force_t* forces, int np, int tsteps) {
    int mpi_rank, mpi_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
    deep_booster_alloc(MPI_COMM_WORLD, mpi_size,1,&workers);

    for (int t = 0; t < tsteps; t++) {
        #pragma omp task in([n_blocks] local) inout([n_blocks] forces, [n_blocks] tmp) onto(workers,mpi_rank)
        {
            int mpi_size;
            MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
            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;
            }
        }

        #pragma omp task inout([n_blocks] local ) inout([n_blocks] forces)
        update_particles(n_blocks, local, forces, time_interval);
    }

    #pragma omp taskwait
    deep_booster_free(MPI_COMM_WORLD, mpi_size,1,&workers);
}
```
5. EVALUATION 5.1 Performance of offloaded code

**Figure 5.1:** Calculate forces particle communications among iterations. Represents how each rank owns a local partition of the particles which will be exchanged with other nodes in order to calculate the accumulated forces for that partition.

calculate forces updates each particle speed and position. These parts are performed on every iteration.

In Figure 5.1 we can see how `calculate_forces` is divided between many MPI processes, so one process calculates partial forces between his own particles, and then exchanges his particles with his neighbours in order to calculate another group of partial forces. After doing this interchange with every other process in the computation, the forces will be fully calculated. Afterwards, particle positions and speeds are calculated on `update_particles`.

In table 5.1 we can see in which devices each part of our benchmark is executed. For the Intel Offload execution, communication has to be performed through the hosts, because as explained before, those programming models do not allow communications between different nodes to be performed inside of kernels.

We can see a comparison of the data communications needed done during the
5. EVALUATION  

5.1 Performance of offloaded code

Table 5.1: NBody Execution Device Schema

<table>
<thead>
<tr>
<th>Programming Model (MPI+)</th>
<th>Setup &amp; check</th>
<th>Calc_forces O(n^2)</th>
<th>exchange_particles O(n), MPI</th>
<th>update_part, O(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native OmpSs(A)</td>
<td>Xeon Phi</td>
<td>Xeon Phi</td>
<td>Xeon Phi</td>
<td>Xeon Phi</td>
</tr>
<tr>
<td>OmpSs Offload(B)</td>
<td>Host</td>
<td>Xeon Phi</td>
<td>Xeon Phi</td>
<td>Xeon Phi</td>
</tr>
<tr>
<td>OmpSs Offload(C)</td>
<td>Host</td>
<td>Xeon Phi</td>
<td>Xeon Phi</td>
<td>Host</td>
</tr>
<tr>
<td>Intel Offload(E)</td>
<td>Host</td>
<td>Xeon Phi</td>
<td>Host</td>
<td>Xeon Phi</td>
</tr>
</tbody>
</table>

Figure 5.2: OmpSs NBody Simulation offload (Single iteration communications). Master processes will spawn two worker processes which can communicate internally in the Xeon Phi and have them calculating the forces. Total Communications = 2N/NP + N

benchmark using our offload and Intel Offload in Figures 5.2 and 5.3.

When implementing this simulation with Intel Offload or OmpSs offload, there are some differences, some of them come because of the fact that Intel Offload is not capable of communicating between the MICs.

By looking at listings 5.1 and 5.2, we can compare offload directives, syntactically both directives look similar, but in OmpSs offload, programmer only has to specify the dependencies/copies, as data allocation and transfers are handled automatically by our runtime instead of having to specify allocate and free on pragmas whenever programmer wants to reuse data like when using Intel Offload. Also in OmpSs Offload user can offload the internal loop as a whole, as remote processes can exchange particles using MPI, in Intel Offload this has to be performed in the local node.
5. EVALUATION

5.1 Performance of offloaded code

Listing 5.2: NBody implemented with Intel Offload

```c
void parallelCalcForces(particle_t* local, particle_t* tmp, force_t* forces, int np, int tsteps) {
    int mpi_rank, mpi_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    for (int t = 0; t < tsteps; t++) {
        particles_block_t * remote = local;
        for(int i=0; i < mpi_size; i++){
            #pragma offload target(mic) in(local[0:n_blocks] :
                alloc_if(t==0) free_if(0)) in(remote[0:n_blocks]
                : free_if(i==rank_size-1) alloc_if(i<=1) out(
                forces[0:n_blocks] : alloc_if(t==0) free_if(0))
            calculate_forces(forces, local, remote, n_blocks);

            exchange_particles(remote, tmp, n_blocks, rank, rank_size , i);
            remote=tmp;
        }

        #pragma offload target(mic) inout(local[0:n_blocks] :
            alloc_if(0) free_if(t==tsteps-1)) inout(forces[0:n_blocks]
            : alloc_if(0) free_if(t==timesteps-1))
        update_particles(n_blocks, local, forces, time_interval);
    }
}
```
5. EVALUATION  

5.1 Performance of offloaded code

As seen in Figure 5.4, single node performance is almost the same when comparing native versions with offload versions, so we can say that there is no performance losses when executing code which was offloaded when comparing to executing code in native mode. Data transfers from master node to offload node will have a cost when offloading, but they are negligible in this case. Apart from this, we can see that, as expected, host execution is much slower than Xeon Phi for this benchmark.

Same results can be seen on Figure 5.5 regarding scalability, in which we see that the performance of the offload implementation is the same than executing the code natively on the MICs. Performance keeps constant from 1 to 128 nodes, so we show that OmpSs offloading does not introduce any particular problem which prevent the offloaded parts of the code to scale and perform as as well as they do if they were executed natively.

We executed our benchmark on Stampede Supercomputer with 128 nodes, from Figure 5.6 we can see results for strong-scaling test. In this comparison, we can compare with Intel Offload implementation, which seems to be performing better than our
5. EVALUATION

5.1 Performance of offloaded code

Figure 5.4: Single node performance of each of the different versions of the NBody Benchmark. We can see how it’s slower in the host and all the other versions, including our offload, obtain similar performance.

Figure 5.5: Weak Scaling Speedup of different versions of NBody Benchmark
5. EVALUATION

5.2 Overhead comparison with native MPI

Figure 5.6: Strong scaling speedup of different versions of NBody Benchmark

implementation when theoretically it should perform worse as we allow direct MIC-to-MIC communications. This is caused by Stampede not having an optimized InfiniBand driver on the MICs (yet), so transferring data between hosts using InfiniBand and then transferring data using QDR bus between host and MIC is actually faster than transferring data between MIC and MIC (which uses Ethernet and goes through the host).

5.2 Overhead comparison with native MPI

Now that we saw that code is not executed slower by just offloading it, we have to compare the performance versus a theoretical native MPI implementation. In order to do this, there are two possible sources of overhead, the one which DEEP Booster Alloc introduces compared with MPI_Comm_Spawn and the one in offloading tasks.

At Figure 5.7 we can see the time taken by each step of the spawn, we can see the overhead of initializing our runtime and offload structures is negligible compared with native MPI_Comm_spawn. In addition to this, scalability is 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 pro-
5. EVALUATION

5.2 Overhead comparison with native MPI

Figure 5.7: OmpSs vs Native MPI Spawn. Each bar shows how time is distributed in MPI Spawn time, which is the time that MPI takes to spawn the processes, and the extra operations we introduce to setup our offload mechanism.

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.

Listing 5.3: Static test OmpSs

```c
#define N_STEPS 500

void sum(int* const __restrict__ vector1, int* const __restrict__ vector2, const int arraySize) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    for (int i=0; i<100000; ++i) vector2[i%arraySize] += rank;
}

int main(int argc, char **argv )
{
    const int ARR_SIZE=10000000;
    int number_of_spawns = 2;
    MPI_Comm workers;
```
5. EVALUATION  5.2 Overhead comparison with native MPI

double start1=wall_time();
deep_booster_alloc(MPI_COMM_WORLD,number_of_spawns,1,&
    workers);
double start=wall_time();
int vector1[ARR_SIZE];
int vector2[ARR_SIZE];
int vector3[ARR_SIZE];
int vector4[ARR_SIZE];
for (int i=0; i<ARR_SIZE; ++i) {
vectord1[i]=0;
vectord2[i]=0;
vectord3[i]=0;
vectord4[i]=0;
}
for (int ns=0; ns<N_STEPS; ns++) {
    #pragma omp task inout(vector1[0;ARR_SIZE],vector2
        [0;ARR_SIZE]) onto(workers,0)
    {
        sum(vector1,vector2,ARR_SIZE);
    }
    #pragma omp task inout(vector3[0;ARR_SIZE],vector4
        [0;ARR_SIZE]) onto(workers,1)
    {
        sum(vector3,vector4,ARR_SIZE);
    }
    #pragma omp task inout(vector1[0;ARR_SIZE],vector2
        [0;ARR_SIZE],vector3[0;ARR_SIZE],vector4[0;
        ARR_SIZE])
    for (int i=0; i<ARR_SIZE; ++i) {
        vectord1[i]+=10;
vectord2[i]+=10;
vectord3[i]+=10;
vectord4[i]+=10;
    }
}
#pragma omp taskwait
double end=wall_time();
printf("Vector values is %d\n",vector1[50]);
printf("OmpSs Total execution time : %g s.\n", end - start);
printf("OmpSs Spawn execution time : %g s.\n", end1 - start1) ;
5. EVALUATION

5.2 Overhead comparison with native MPI

deep_booster_free(&workers);

MPI_Finalize();

Listing 5.4: Static test native MPI

#define N_STEPS 500
#define ARR_SIZE 10000000

void sum(int* const __restrict__ vector1, int* const __restrict__ vector2, const int arraySize) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    for (int i=0; i<100000; ++i) vector2[i%arraySize]+=rank;
}

int main( int argc, char **argv )
{
    if (argc>1){
        int provided;
        MPI_Init_thread(0,0,MPI_THREAD_MULTIPLE,&provided);
        MPI_Comm parent;
        int arraySize;
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_get_parent(&parent);
        MPI_Recv(&arraySize,1,MPI_INT,0,100,parent,MPI_STATUS_IGNORE);
        int* vector1=malloc(sizeof(int)*arraySize);
        int* vector2=malloc(sizeof(int)*arraySize);
        //For N_STEPS, recv, add, send
        for (int ns=0; ns<N_STEPS; ns++) {
            MPI_Recv(vector1,arraySize,MPI_INT,0,100,parent,MPI_STATUS_IGNORE);
            MPI_Recv(vector2,arraySize,MPI_INT,0,100,parent,MPI_STATUS_IGNORE);
        }
    }
}
5. EVALUATION

5.2 Overhead comparison with native MPI

```c
sum(vector1, vector2, arraySize);

MPI_Send(vector1, arraySize, MPI_INT, 0, 100, parent);
MPI_Send(vector2, arraySize, MPI_INT, 0, 100, parent);
}
free(vector1);
free(vector2);
MPI_Finalize();
return 0;
}

int number_of_spawns = 2;
MPI_Comm workers;
MPI_Status status;
int size, again;
int provided;
MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
MPI_Comm_size(MPI_COMM_WORLD, &size);
char *array_of_commands[number_of_spawns];
char **array_of_argv[number_of_spawns];
MPI_Info array_of_info[number_of_spawns];
int n_process[number_of_spawns];
int i = 0;
for (i = 0; i < number_of_spawns; i++) {
    n_process[i] = 1;
    char *argvv[] = {"dummyarg", NULL};
    array_of_argv[i] = argvv;
    MPI_Info info;
    MPI_Info_create(&info);
    MPI_Info_set(info, "host", "mic0"); // Set MIC ip address here
    array_of_commands[i] = "/vecsum.mic";
    array_of_info[i] = info;
}
double start1 = wall_time();
MPI_Comm_spawn_multiple(number_of_spawns, array_of_commands, array_of_argv, n_process, array_of_info, 0, MPI_COMM_WORLD, &workers, MPI_ERRCODES_IGNORE);
```
double end1=wall_time();
double start=wall_time();
int vector1[ARR_SIZE];
int vector2[ARR_SIZE];
int vector3[ARR_SIZE];
int vector4[ARR_SIZE];
int arraySize=ARR_SIZE;
for (int i=0; i<ARR_SIZE; ++i) {
    vector1[i]=0;
    vector2[i]=0;
    vector3[i]=0;
    vector4[i]=0;
}
MPI_Send(&arraySize,1,MPI_INT,0,100,workers);
MPI_Send(&arraySize,1,MPI_INT,1,100,workers);

//For N_STEPS, send recv and add
for (int ns=0; ns<N_STEPS; ns++) {
    MPI_Send(&vector1,ARR_SIZE,MPI_INT,0,100,workers);
    MPI_Send(&vector2,ARR_SIZE,MPI_INT,0,100,workers);
    MPI_Send(&vector3,ARR_SIZE,MPI_INT,1,100,workers);
    MPI_Send(&vector4,ARR_SIZE,MPI_INT,1,100,workers);

    MPI_Recv(&vector1,ARR_SIZE,MPI_INT,0,100,workers,
              MPI_STATUS_IGNORE);
    MPI_Recv(&vector2,ARR_SIZE,MPI_INT,0,100,workers,
              MPI_STATUS_IGNORE);
    MPI_Recv(&vector3,ARR_SIZE,MPI_INT,1,100,workers,
              MPI_STATUS_IGNORE);
    MPI_Recv(&vector4,ARR_SIZE,MPI_INT,1,100,workers,
              MPI_STATUS_IGNORE);

    for (int i=0; i<ARR_SIZE; ++i) {
        vector1[i]+=10;
        vector2[i]+=10;
        vector3[i]+=10;
        vector4[i]+=10;
    }
}

double end=wall_time();
printf("Vector values is %d\n",vector1[50]);
As seen in Listings 5.4 and 5.3 we implemented a very simple program where all native MPI tasks are hard-coded, something which in complex programs where the control flow is more complex and depends on decisions taken in the host will not be easy to code. Even in this simple program, in addition to differences in logical complexity, the amount of useful lines of code is 44 in OmpSs and 100 in native MPI.

At Figure 5.8 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.

### 5.3 Real applications

Our objective with this offload model is not only to get as good performance as possible when offloading, but allowing users to offload their software without much effort.
In order to do this, three external applications have been ported to our offload system. These applications are part of the DEEP/DEEP-ER project and are still being improved. These applications implement a master-slave model and almost full offload (I/O on hosts and everything else on Phis), which is one of the ways to use our model.

### 5.3.1 SRMIP

This C application does seismic imaging and follows a master-slave pattern, as the code of the application is not public, we ported a prototype provided by the owners of the code, in this prototype MPI processes are split into groups, being the first rank in each group the master of the group. This approach works correctly in single architecture machines, but in multi architecture machines, there may be a problem as programmer will have to control where he wants to execute each one of the workers.

In each group work comes from the master, as seen in Listing 5.9, which sends a work order to the slaves and waits until the slaves finish processing the work. Being this operation bound to the amount of processes, so each master has the same amount of workers. After every worker finishes, masters merge the results of all the groups into a single result.

With our offload model, only masters will be launched in the application, as seen in Figure 5.10, and each master will allocate as many workers as needed, so having a
dynamic amount of workers per master. User can get a dynamic amount of workers per master without all the hassle which comes from managing which ranks are workers and masters in each group. In addition to this, workers can be seen by every master (this is optional, depending on the allocation strategy), so different masters can use the pool of workers globally, allowing to perform load balancing between different masters.

The resulting pseudo-code of the application can be seen in 5.5.

As explained before, the structure of this application fits our programming model very well, so apart from offloading tasks we were able to hide most of the complexity of implementing a master-slave mechanism inside MPI, so only 4 MPI Calls (communication between the masters) are needed in our version, the number of lines of code needed which can be seen in Table 5.2:
### Listing 5.5: SRMIP offload pseudo-code

```c
void main(int argc, char* argv[]) {
    MPI_Comm_size(MPI_COMM_WORLD, &masters);
    // Allocate and init image_t data structure
    image_t global_pool, master_image, worker_pool[workers];
    MPI_Comm comm_workers;
    DEEP_Booster_alloc(MPI_COMM_WORLD, n_workersm, 1, &
                        comm_workers);
    int my_jobs = jobs / masters;
    for(int i=0; i<my_jobs; i++) {
        int idx = i % workers;
        #pragma omp task out(worker_pool[idx]) onto( 
             comm_workers)
        worker(worker_pool[idx]);
        #pragma omp task inout(master_image) in(worker_pool 
            [idx]);
        accum(master_image, worker_pool[idx])
    }
    #pragma omp taskwait
    mpi_global_accum(global_pool, master_image);
}
```

5. EVALUATION 5.3 Real applications
5. EVALUATION

5.3 Real applications

Table 5.2: Comparison of the number of code lines with and without OmpSs offload

<table>
<thead>
<tr>
<th>SRMIP results</th>
<th>Original</th>
<th>OmpSs Offload</th>
</tr>
</thead>
<tbody>
<tr>
<td># Code Lines</td>
<td>415</td>
<td>314</td>
</tr>
<tr>
<td># MPI Calls</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td># OmpSs Pragmas</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

5.3.2 TurboRVB

This Fortran application is a MPI based application where every process performs an initialisation which reads data from files and performs some MPI communications, after doing so, it performs the computation and writes the output. It works well on the MICs, so the objective with this application was only to execute it completely in the MICs without having to rely if files were available inside the MIC file-system.

In order to offload this application, the initialization phase was split in two different phases, a local initialization phase where data from files is read and some of the input parameters are allocated. After doing so, each process will allocate one remote MIC node and the rest of the computation will be offloaded to that node. Once the offload begins, a post-initialization phase is performed, and all temporary arrays are allocated in remote nodes with the parameters read in the cluster part of the execution.

The main challenge when offloading this application came from their extensive use of global/module variables inside the offloaded region, as according to OpenMP standard, variables which are private for a task are only private for the task region, but not for calls to external functions in different modules which use the same variable. This restriction which only applies for real single memory space programs made offloading big codes very difficult. As we had implemented an offload model in which we have another instance of the process in which global variables are somehow private, in the sense that are not used by anyone except tasks and do not share value with the main instance of the program, we can modify them without affecting the ones in the main program, so whenever we offload a task which uses global variables, we copy the value of the private variable into the region pointed by the global variable, and make the private variable point the global variable. This way, both the task code and the calls to functions which are performed inside the task code, will use the same variable.
5.3.3 FWI

Full Wave Inversion is an oil related application which has been developed by the CASE department of BSC.

A very simplified pseudo-code of this application can be seen at listing 5.6, we can see that this application has a master process (level 1) and two levels of Offload (levels 2 and 3). The master process will allocate the first level of offloads and afterwards each worker of this first level will allocate its own second level offload workers. Once all the nodes have been allocated, they are ready to execute all the tasks compose the application.

First the master will offload one shot to process to each of the first level workers, which will decompose this shot in as many parts as needed and send the shot to the second-level workers so they process the shot. After all of this work has been finished and all shots have been processed, a similar workflow will be executed in order to merge these shots.

By looking at figure 5.11 we can see a trace obtained with Extrae of a short execution of this program with 12 shots and a distribution of 1x4x4 (17 total nodes) workers. We can see how first the second level boosters are allocated in the first part of the program and how after doing so, second level workers will decompose the shot and send
void main(int argc, char* argv[]) {
    MPI_Comm_size(MPI_COMM_WORLD, &masters);
    // Allocate slaves
    DEEP_Booster_alloc(MPI_COMM_SELF, n_slaves, &comm_slaves);
    for(int i=0; i<n_slaves; i++) {
        // Order slaves to allocate workers
        #pragma omp task onto(comm_slaves, i)
        DEEP_Booster_alloc(MPI_COMM_SELF, n_workers, &comm_workers);
    }
    for(int i=0; i<nshots; i++) {
        #pragma omp task out(shots[i]) onto(comm_slaves)
        processShot(shot[i]);
    }
    for(int i=0; i<nshots; i++) {
        #pragma omp task in(shots[i]) inout(merger) onto(comm_slaves)
        merge_shot(shots[i], merger);
    }
}

void processShot(int shotID) {
    double decomposedShot[n_workers]=readAndDecShot(shotID);
    for(int i=0; i<n_workers; i++) {
        #pragma omp task inout(decomposedShot[i]) onto(comm_workers, i)
        processDecomposedShot(decomposedShot[i]);
    }
}
5. EVALUATION

5.3 Real applications

Figure 5.12: FWI execution 2. Shows how each group of 4 level 3 nodes process different shots at the same time independently.

the computationally intensive part of the algorithm to be executed at the third level of the offload. At figure 5.12 we can see a zoomed view of how this computational part of the algorithm is executed in 4 different groups of 4 workers each which perform MPI communications internally.
Chapter 6

Conclusions and Future Work

This master thesis shows how OmpSs, a programming model that runs sequentially written applications following a task based data-flow execution model, 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 amount of tasks per second which can be offloaded is very high. 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. We believe our OmpSs offloading capabilities will help to exploit current and future heterogeneous clusters, providing application developers an effective tool to code complex applications with a MPMP execution model that really fits the underlying hardware.

There are several areas of future work that we plan to work on, like integrating our offload technique with existing job managers so users do not need to specify destination nodes manually with host files or extending the uses of this technique to facilitate migration and malleability of MPI applications.
References


