Abstract—Developing complex scientific applications on high performance systems requires both domain knowledge and expertise in parallel and distributed programming models. In addition, modern high performance systems are heterogeneous, thus composed of multicores and accelerators, which despite being efficient and powerful, are harder to program. Domain-Specific Languages (DSLs) are a promising approach to hide the complexity of HPC systems and boost programmer’s productivity. However, the huge cost and complexity of implementing efficient and scalable DSLs on HPC systems is hindering its adoption for most domains. Addressing such problems, we present Data Flow Language (DFL), a DSL designed to exploit distributed and heterogeneous HPC systems. DFL abstracts the key concepts such systems as SMP tasks for multicores, kernels for accelerators and high-level operations for distributed computing. In addition, DFL leverages the hybrid MPI/OmpSs data-flow programming model to efficiently implement the previous concepts. All of these features make DFL suitable as the target language for other DSLs. However, it is also suitable as a fast prototyping language to develop distributed applications on heterogeneous systems.

Keywords—DFL, DSL, OmpSs, HPC, MPI, LMS, Scala.

I. INTRODUCTION

In the emerging field of domain-specific languages for HPC, domain experts such as physicists, geologists and biologists, solve their problems using a DSL, which hides all the complex and low-level details of programming an HPC system. However, the implementation of a DSL that is both efficient and scalable on HPC systems is a challenging task that requires a lot of effort, limiting the feasibility of the DSL approach to few relevant or widely-used domains such as partial differential equations [25] or molecular dynamics [3]. The implementation cost of custom DSL compilers and runtime systems for most domains is very hard to amortize due to its very specific and limited nature. The goal of this work is to create a framework that eases the design and implementation of DSLs for HPC environments. To that end, we have developed Data Flow Language (DFL), a DSL that abstracts the key concepts needed to implement an efficient HPC application. DFL is embedded in Scala [16] using Lightweight Modular Staging (LMS) [18] and it uses the hybrid MPI/OmpSs [9] programming model as the target platform to generate efficient distributed applications that can exploit heterogeneous HPC systems.

DFL provides a dataflow model based on four concepts: Buffers, tasks, kernels and high-level operations. Buffers abstract the concept of data, while tasks and kernels represent computations written in C++ and OpenCL on a multicore or accelerator, respectively. These buffers, tasks and kernels have a one-to-one relation with the data, tasks and kernels concepts of OmpSs. Only with them, DFL already provides a flexible and powerful platform to fully exploit single node performance. Additionally, DFL uses high-level operations such as map, reduce, divide and conquer, etc. to exploit distributed systems without exposing a low-level message passing interface such as MPI. Each high-level operation is a computational pattern that describes how to apply a computational task or kernel over a set of input data. In particular, the contributions of this paper are the following:

- DFL, a DSL that abstracts the hybrid MPI/OmpSs programming model using buffers, tasks, kernels and high-level operations.
- Abstraction of OpenCL C programs (kernels) as objects, easily composable in a single application.
- Transparent and efficient memory management.
- Interaction between host and accelerator code checked at compilation time.
- Full compatibility with existing C++ libraries to maximize productivity and leverage existing libraries.
- Efficient and scalable implementation of Divide and Conquer as a high-level operation to exploit clusters and distributed systems.

The rest of this document is organized as follows. First, Sec. II describes the main technologies DFL is built upon. Next, Sec. III reviews the core features of DFL. Then, Sec. IV describes in detail the most complete high-level operation of DFL, which is divide and conquer. Following on, Sec. V provides a complete DFL example using the divide and conquer components to generate MPI code for a simple application. Then, Sec. VI reports our experience using DFL to implement powerful HPC DSLs for heterogeneous architectures. The document continues with Sec. VII with references to related work and, lastly, Sec. VIII gives an overall review of our current lines of research involving DFL.

Barcelona Supercomputing Center

September 26, 2014
II. UNDERLYING TECHNOLOGIES

DFL is built on top of two main software technologies. The first is LMS, a Scala library for embedding DSL compilers together with DSL applications, thus reusing all of the Scala features to define new languages. The other key pillar of DFL is OmpSs, a parallel task-based programming model that is composed of pragma extensions for C/C++/Fortran, the Mercurium compiler and the Nanos++ runtime library. By translating DFL applications into C++ OmpSs/OpenCL, key decisions like platform/device(s) selection, memory transfers between host and accelerators and task scheduling are deferred to Nanos++, which is the specialized HPC runtime system by BSC. Our objective with DFL is to create an infrastructure that can be reused by many DSLs. At the same time, we are using LMS to reuse the compilation infrastructure of the DSL implementations and MPI/OmpSs to reuse the execution runtime DFL is built upon. The rest of this section provides a quick overview of the main Scala features used throughout this document, as well as details on LMS and OmpSs.

A. Scala

Scala is a statically typed, multi–paradigm programming language with type inference. It supports both functional (functions as first-class citizens) and object-oriented (values are objects, mixin composition with traits) programming paradigms. It also supports concurrency with the actor model.

1) Objects and classes: Scala provides classes, objects and traits, which are used to implement data types. First, classes are the standard well known state+behavior encapsulation unit. Then, objects are singleton, static containers. If a class and an object share the same name, they are called companion classes/objects, which is the way Scala uses to split the static and the dynamic parts of classes. Traits are meant to encapsulate features attachable to classes and even other traits, which is the way Scala deals with multi-modular composition without multiple inheritance.

2) Traits and mixin composition: Traits in Scala are the basic unit of code reuse. Traits are declared in a similar way as regular classes with the exception that the Scala programmer uses the keyword trait instead of class. Scala classes can mix in several traits using the keywords extends and with. By mixing in several traits into classes, Scala provides stackable in-order composition.

3) Generics: Scala supports type parametrization by introducing the concept of generic classes, which allow the programmer to define types that can operate in a type-safe manner on values without relying on their types:

```scala
class TypePrinter[T:Manifest] {
  def print() = m.toString
}
```

In this example, we can see the declaration of a generic class TypePrinter[\_], which is ready for some unspecified type T. Generics implement parametric polymorphism: implementation and interface of generic TypePrinter[\_] is not coupled with any concrete type such as Int or String. This coupling is achieved when the generic type is instantiated and a type is passed as parameter to the generic class.

4) Pattern matching: One of the functional features that Scala implements is pattern matching. Scala allows the programmer to match values of any type with a match-first policy. However, Scala, as an object-oriented language, extends this concept to objects:

```scala
def findRoom(no: Int): String = no match {
  case 103 => "Lab"
  case 105 => "Dean"
  case 104 => "Secretary"
  case _ => "Empty"
}
```

Function findRoom takes an Int and pattern-matches it against several cases. If no is equal to one of these values, the corresponding string is returned. The statement case _ if reached, always yields True and acts as the “default case”.

5) Abstract type members and implicit: In Section II-A1 we mentioned that types can be abstract members of a class. This is another way of building abstractions in Scala (in addition to type parametrization, described briefly in Section II-A3). Abstract type members, as was the case with generics and type parametrization, allow programmers to abstract types over both interfaces and implementations. This example implements the same functionality as the previous TypePrinter using abstract type members:

```scala
trait TypePrinter {
  type T
  implicit val m: Manifest[T]
  def print() = m.toString
}
```

First, we define a trait that contains an abstract type member, which will be defined in classes that mix-in the TypePrinter trait. Then, we initialize the m value with the implicitly keyword, which retrieves an implicit value whenever possible (if there is any in scope). Afterwards, the type member T is defined in classes Integrals and Doubles by simply assigning a concrete type to it. Calls to method print give the same results as in the generics example.

B. LMS

Lightweight Modular Staging (LMS) [18] is a technique for embedding DSLs as libraries into Scala as a host language, while enabling domain-specific optimizations and code generation. LMS uses types to distinguish between DSL application code and regular Scala code. Expressions belonging to the DSL application are called “staged expressions” and have type Rep[T] when they correspond to a computation yielding a T in the DSL application. When the DFL compiler and an application are compiled together with the Scala Virtualized Compiler\(^1\) [15], they result in a Scala application. When this

\(^1\)The Scala Virtualized compiler is a special version of the Scala compiler required by LMS in order to redefine the behavior of the core Scala constructs.
Scalable application run, the DSL application is potentially optimized and translated to the next stage, normally a lower level source code, although it is possible to implement the DSL compiler so it produces a final binary.

The definition of a DSL in LMS consists of three main parts, the DSL interface, its internal representation, and the code generation. All three parts can be made of several modular components and mixed-in together using Scala traits. This way, DSL users write their programs in terms of the DSL interface only, allowing the implementation to be hidden. To see how to define a DSL with LMS, consider the following example:

```scala
trait DSLOps extends Base {
  def sayHelloTo(name: Rep[String]) : Rep[Unit]
}
trait DSLOpsExp extends DSLOps with EffectExp {
  case class SayHelloTo(name: Exp[String])
  def sayHelloTo(name: Exp[String]) = reflectEffect(SayHelloTo(name))
}
trait DLSCodeGen extends CGenEffect {
  val IR: DSLOpsExp
  import IR._
  override def emitNode(sym: Sym[Any], rhs: Def[Any]) =
    rhs match {
      case SayHelloTo(s) =>
        stream.println("\nHello, %s!\n", * quote(s) + ")")
      case _ => super.emitNode(sym, rhs)
    }
}
```

DSL types are wrapped with the abstract type constructor `Rep[T]`, where T is a DSL type. As shown in section II-A, abstract types are types whose identity is not precisely known. This way, one can define generic containers of type T that will be instantiatied to the particular type. In the example, no new types are defined, so Rep is only used to specify the type of the `name` parameter in line 2. In the implementation of the function in line 7, `Rep[T]` types become `Exp[T]`, which stands for the implementation type. Next, `Exp` types correspond to Intermediate Representation (IR) nodes by means of two different subclasses: The first one is `Const[T]`, for constants; and the other is `Sym[T]`, for named values defining a `Def[T]`. `Def[T]` is just the base class for all IR nodes represented by `Exp[T]` types. Again, since our example defines no DSL types, the IR node `SayHelloTo` in line 5 is just a computation that yields `Unit` as a result. Lines 12 and 13 bring into scope new types, the IR node `SayHelloTo` and `Def[T]`. Afterwards, the `emitNode` function in line 13 handles how to generate code for the only IR node of the DSL and defers the generation of any other type of node to the generic C code generator (line 18). In Scala, implicit parameters like the `stream` in line 14 are just a convenient way of taking an instance of `PrintWriter` from the caller’s scope without it being specified explicitly. This is useful for recursive functions in order to avoid explicit “forwarding” of some parameters and make the code look cleaner. Finally, notice the use of the `reflectEffect` function in line 7 to tell LMS that the IR node produces a side effect on the system (prints a message to a stream), so LMS can internally schedule the code generation accordingly.

C. OmpSs

OmpSs is a high-level, task-based, parallel programming model supporting SMPs, heterogeneous systems and clusters. It extends OpenMP task `pragma`s with constructs for specifying data dependencies between tasks and the devices they will run in. The OmpSs programming model consists of a source-to-source translator called Mercurium, and a runtime library, Nanos++. First, Mercurium translates Fortran, C or C++ code with OmpSs pragma annotations to regular code with calls to Nanos++, which delimit regions of code or functions that will be executed asynchronously. When an application is executed, Nanos++ handles the efficient parallel execution of tasks. It is worth noting that the OmpSs program does not require any call to the OpenCL API [19] because the Mercurium compiler will generate all the required code to setup the kernel, while the Nanos++ runtime maintains a real-time dependency graph and performs the required data transfers from the host to the accelerator and vice-versa, finally running the kernel in one or many SMP/OpenCL/CUDA devices, depending on the pragma options specified. Notice that SMP tasks are tasks meant to run on the CPU threads, whereas OpenCL and CUDA tasks are run on accelerators. The most relevant features of OmpSs used by DFL are host-side tasks, pure parallel loops and device tasks. The first one, host side tasks, are asynchronous function calls that run on the host side of the application (SMT processors). A simple example of OmpSs code producing a host-side task is:

```scala
#pragma omp task inout(a) in(b)
void sum(int &a, int &b) { a += b; }
```

In this example, each time `sum` is called, Nanos++ adds it to the current dependency graph and executes it as soon as its dependencies are met. The following main OmpSs feature that DFL uses is pure parallel loops. As shown in section III-A, buffers offer the capability of mapping a function over each of their elements. As pure function mapping is by nature a full parallel operation, it can be performed in a parallel loop. To run a loop in parallel in OmpSs, it is just required to specify the corresponding pragma, as shown in the following example:

```scala
#pragma omp for for (unsigned i = 0; i < N; i++) a[i] += b[i];
```

Finally, device tasks are the last important feature of OmpSs used by DFL. In OmpSs, device tasks can run on SMT, CUDA or OpenCL devices, but since DFL is currently aimed at OpenCL devices as justified in the introduction, only OpenCL device tasks are used. In addition, OmpSs poses no restriction on where the OpenCL kernel is, so it can be placed together with the host application as long as the code region is marked by the corresponding OpenCL device task pragma. An example of an OpenCL device task in OmpSs is:

```scala
unsigned N = 1024;
#pragma omp device(opencl) nrange(1, N+10, N)
#pragma omp task inout([N+10] a) in([N+10] b)
_kernel void sum([global int *a, global int *b) {
  unsigned id = get_global_id(0);
  a[id] += b[id];
}
```

It is interesting to notice that any host-side constant in scope can be captured and used to specify the parameters of the
kernel execution, thus adding value to the fact that kernels can be written together with the host application. However, DFL applications work with separate OpenCL C kernel files to keep DFL (Scala) code independent from the accelerator code while providing safety mechanisms to guarantee a correct interaction. These safety mechanisms are implemented via an special stage in Mercurium (the OmpSs compiler). When the DFL compiler is executed, Mercurium is invoked with the OpenCL C files used by the DFL program. Then, Mercurium parses them and returns the type information so the DFL compiler can type check all function calls.

III. DFL FEATURES

Inspired by the OmpSs [9] programming model, DFL provides high-level constructs with the most common terms of heterogeneous programming like kernels, data buffers and grids (N-Dimensional Ranges or NDRanges, for short). With these features DFL provides a powerful abstraction to implement HPC DSLs that run on machines composed of CPUs and accelerators. To begin with, this section highlights the most special features of DFL related to heterogeneous programming.

A. Data Flow Design

In the majority of HPC applications, the emphasis is on large amounts of data being processed in parallel, thus the key concept of heterogeneous application design is how data flows between computing devices. The core syntax of DFL follows this data flow nature. First of all, there is a data buffer abstraction, which represents a regular data container. In DFL, Buffer objects represent plain data containers with automatically managed allocation and usage. Thanks to the expressive power of Scala, buffers provide type inference when initialized with immediate values. A buffer can be just allocated, by specifying its size and element type as follows:

```scala
// Allocate space for 4096 floating point values
val buf1 = Buffer.fill[Float](4096)
// Buffer of 4 integer values
val buf2 = Buffer(5, 8, 21, -3)
```

DFL buffers support one of the most common operations over a Scala list: map. This operation applies a particular function to a list and returns the new list resulting from it. In the case of DFL buffers, the map semantics are not exactly the same, since the function does not return a new buffer, but rather updates the contents of it with the result of applying a given function to all of its elements. One of the simplest use cases of mapping a function to a buffer would be to initialize all elements with random values², as the following example shows:

```scala
val myData = Buffer.alloc[Float](1024)
myData map ( _ ⇒ rand)
// myData now contains 1024 random floating point values
```

B. Tasks and Kernels

Tasks on the host side and accelerator kernels both process the data in the buffers. To begin with, a kernel container represents a file that contains kernels written in OpenCL C. Assume we have this simple OpenCL C program:

```opencl
kernel void add( global float* a, 
                global float* b, 
                global float* c) { 
  unsigned int i = get_global_id(0); 
  c[i] = a[i] + b[i]; 
}
kernlel void sub(global float* a, global float* b) { 
  unsigned int i = get_global_id(0); 
  a[i] -= b[i]; 
}
```

Then, a kernel container is simply declared like this:

```python
val kc = KernelContainer("/path/to/kernels.cl")
```

Once a kernel container is declared, kernels can be retrieved from it using the `Kernel` construct. For example, given the declaration of `kc` in the previous example, it can be used like this:

```python
// This retrieves the 3-parameter "add" kernel 
// from kc, namely myAdd
val myAdd = Kernel(kc, "add")(In, In, Out)
// Put on mySub the 2-parameter kernel named
// "sub" inside kc
val mySub = Kernel(kc, "sub")(InOut, In)
```

Notice the second list of arguments given to the `Kernel` construct, where the directionality of each argument inside the kernel body is specified in order for the scheduler to properly insert this task in the dependency graph. In addition, DFL ensures that all kernel calls match their corresponding prototypes in the containers, so any error like a typographic error on the kernel name or a non-existing kernel will be signaled by the DFL implementation.

To determine how a kernel runs, DFL uses NDRanges just in the way OpenCL does, but adds the possibility to store NDRanges as objects so they can be reused and composed without significant effort. As shown in the following example, NDRanges of two or more dimensions are composed of one dimensional ones, providing a simple and convenient way of declaring the parallelization scheme of OpenCL tasks.

```scala
// Block size = 64, Number of blocks = 32
val x = Range1D(64, 32)
val xy = Range2D(x, x)
val xyz = Range3D(x, y, Range1D(128, 8), x)
```

Of course, the NDRanges can be directly specified in line with the kernel call just like in regular OpenCL code, but this approach provides a more object-oriented mechanism. Once the kernel instances and the NDRanges have been created, kernels can be called with actual parameters:

```scala
val range1 = Range1D(4, 1)
val input1 = Buffer(4, 1, 2, 5)
val input2 = Buffer(2, 6, -2, 9)
val out = Buffer.fill[Int](4)
val kc = KernelContainer("kernels.cl")
val addk = Kernel(kc, "add")(In, In, Out)
addk(input1, input2, out) using range1
```

Again, any mismatch in the actual parameters of the kernel calls will trigger an error in the DFL compiler, conveniently shown to the user before the application is run.

²`rand` is a built-in function of DFL.
CPU's can also perform runtime-managed computations using host-side tasks that synchronize with DFL OpenCL kernels. DFL host-side tasks work exactly like OmpSs tasks and they look very similar. For instance, a K-Means algorithm implementation could compute the new centroids of each iteration on the host after the accelerators compute the sum of the points on each cluster. First, the kernel closestSum computes sumPoints and nPoints, which are buffers of 4*K and N elements, respectively. Then, the inline task takes them as input and computes the new centroids. Notice that by specifying the directionality of the arguments, OmpSs is able to synchronize all the necessary memory transfers and dependencies (see section II-C):

```scala
(0 until MAX_ITER) foreach { _ =>
  closestSum(p, nPoints, assig, centroids, sumPoints)
  using range1D
  Task(centroids, sumPoints, nPoints)(Out(4 * K), In(4 * K), In(N)) {
    (0 until K) foreach { i =>
      (0 until 2) foreach { j =>
        centroids(4 * i + j) = sumPoints(4 * i + j) / nPoints(i)
      }
    }
  }
}
```

### C. High level operations

In DFL, high-level operations are used to apply a number of tasks/kernels over a set of data buffers following a specific algorithmic pattern. Some common algorithmic patterns in HPC are map, reduce or even divide and conquer. For example, the map pattern consists of applying a function to all the elements of a data buffer independently of their position in the container. Therefore, it’s a perfectly parallel operation. Another parallel operation is reduce, which consists in applying a combination operation to all the elements of a buffer recursively until all of the elements have been combined. Divide and conquer is a more complex parallel operation, since it involves a sequence of operations and can be considered a complete algorithmic pattern. For this reason, we decided to first provide support for divide and conquer before handling more simple patterns like map and reduce that can be easily considered specific cases of divide and conquer. The high-level operations related to divide and conquer will be explained in detail in Section IV.

### D. Miscellaneous

DFL also provides mechanisms to reuse C/C++ libraries. I/O, and the most common low level features of HPC programming models. In HPC, reusing already existing libraries is quite common, as large-scale software often relies on multiple libraries supported by third party developers. Common examples of external libraries used in C/C++ HPC applications are FFT [11], BLAS [7], LAPACK [21] for high performance math or VTK [2] for visualization. Libraries written for C or C++ are fully compatible with DFL applications.

The following example illustrates how external libraries and IO operations are used within DFL:

```scala
// External includes
include("atlas/clapack", true)
lib("lapack_atlas")

// Defining external functions
def spotrf = LibFun("clapack_spotrf")
// Defining External Symbols
def ColMajor = LibSym[Int]("CblasColMajor")
def Lower = LibSym[Int]("CblasLower")

// Reading an Int from the standard input stream
val N = stdin.get[Int]
// Allocating N*N matrix
val A = Buffer.alloc[Float](N * N)
spotrf(ColMajor, Lower, N, A, N)
```

Let us discuss the code snippet above line by line:

- Line 2 tells the code generator to include atlas/clapack.h to the generated code. The second argument is set to true because this header is written for C and it needs to be wrapped with an extern 'C' block.
- Line 3 tells the linker to add liblapack_atlas to the linking stage.
- Line 6 defines a function that is guaranteed to be available at link time.
- Lines 9 and 10 define symbols that will also be available at link time.
- Line 13 uses all the previously defined external symbols and the buffer created in line 12.

By seamlessly integrating external libraries with high-level code, DFL adds yet another productivity feature for HPC programming environments, as well as allowing DSL compilers targeting DFL to use their already existing C/C++ support libraries.

### IV. Divide and Conquer with DFL

Divide and conquer is a well-known algorithmic pattern and DFL provides a mechanism to bind it to the MPI library. First, each MPI domain is initialized with its corresponding part of the problem’s data. Then comes the divide step of the algorithm, in which a problem is partitioned into smaller subproblems. Following on, each of these subproblems is divided into even smaller subproblems, using all of the computing resources for the process like GPUs and CPUs and the core features of DFL (see section III) to solve them. Once each subproblem is solved, the conquer stage of the algorithm begins. In this stage, each process combines its local solutions, thus obtaining the solution for the whole local process. Finally, all the processes combine their results to get the solution of the whole original problem. This workflow is illustrated in Fig. 1.

To specify a divide and conquer algorithm execution in DFL, four components are required:

- **Domain Allocator**: The MPI domain allocator contains the information on the allocation policy the algorithm requires, which depends on the memory requirements of the particular algorithm.
- **Divider**: The divider is the component in charge of partitioning the problem in a particular format.
Fig. 1. Divide and Conquer workflow of DFL.

- **Combiner**: Local solutions inside the same process are combined by this component. Moreover, the MPI communication pattern required to combine the different MPI domains is also defined by the combiner.

- **Problem**: A problem encapsulates the information related to solving, creating subproblems and initializing the data before starting the algorithm.

The idea behind this decomposition into different components is that the DFL implementation can provide default implementations of the most common kind of domain allocators, dividers and combiners, leaving only the **Problem** component left to be defined for each particular algorithm in most of the cases.

### A. Domain Allocators

Domain allocators are components aimed at specifying how the original problem’s data domain is partitioned among all of the MPI processes. Given the rank (MPI identifier), the domain size (number of MPI processes) and the total size of the problem, the allocator has to determine how much memory is required for each particular process. For example:

```scala
val domAlloc = new LinearAlloc[Int](size=1024)
domAlloc allocDomain(rank=0, nDomains=4)
```

The value `domAlloc` contains an allocator that will split the 1024 elements of the problem across all the processes in a linear fashion, meaning it will assign a continuous range of elements to each process. Therefore, since the process with rank 0 is the first one and there are 4 processes in total, it will allocate space for the first 256 `Float` values. If the problem size is not divisible among the number of processes, the processes with lowest ranks will take one additional element.

DFL provides some default domain allocators for the most common cases, but new allocators can be defined by deriving the corresponding abstract class:

```scala
class MyAlloc[T](val size: Int) extends DomainAlloc[T] {
  override def allocDomain(rank: Int, nDomains: Int) = {
    super.domainSize = size / (rank*nDomains+4)
    super.domainData = new T[super.domainSize]
    super.problemData = domainData
    super.problemSize = domainSize
  }
}
```

Notice that when the domain is allocated, the problem’s data and size match the ones of the domain.

### B. Dividers

In DFL, a **Divider** is characterized by the `divide` method, which given a data structure and its size, splits it and creates the corresponding subproblems. First, one of the most common and natural ways to split a data structure is by taking contiguous chunks of its data and assigning one subproblem to each. This common behavior is implemented by `LinearDivider`, one of the common dividers of DFL:

```scala
val divider = new LinearDivider(baseCaseSize=16)
if (divider.isDivisible(prob))
  val subProbs = divider divide prob
  // subProbs is of type List[Problem[Float, Float]]
```

Here, `prob` is assumed to be a particular implementation of the ```Problem``` abstract class (see Sec. IV-D). Again, if none of the default dividers is suitable for your algorithm, a custom divider can be defined by overriding the `divide` method of the abstract `Divider` class, like in Sec. IV-A with domain allocators.

### C. Combiners

The DFL **Combiner** is the component in charge of combining the local results inside the same process and also communicating with the other processes using MPI to get the solution to the original problem. First, let us take the simple...
case in which the solution to a problem is just a scalar value. The default DFL combiner for that case is ScalarCombiner, which is used as follows:

```scala
subProbs map { prob => prob solve } val combiner = new ScalarCombiner(Float)(mpiComm) val domainSolution : Float = combiner combine subProbs combiner.combineDomains(rank, nDomains)(domainSolution)
```

In this example, mpiComm is the instance of the MPI communicator used. DFL provides an interface to the C++ Boost MPI API [12], in which communicators are instances of mpi::communicator and the MPI collective and point-to-point communications are methods of this class. Similar to what happens with dividers and allocators, custom combiners can be defined to override the combine and combineDomains methods of the abstract Combiner class.

D. Problems

Finally, problems that are solvable with the DFL divide and conquer components are subclasses of Problem. This component is the only one that will be specifically crafted for each problem, because it is where the framework gets the information on how to solve particular instances of the problem itself. The rest of this subsection explains the three functions that characterize a problem solvable with the DFL divide and conquer components, and everything will be put together in the next section with a simple but complete and illustrative example in Sec. V.

1) initDomain: The first function of the Problem interface is initDomain, which, once the allocator has determined how much data this process is going to handle, initializes it by reading the corresponding input files, generating random data, etc.

2) createSubProblem: The next function is createSubProblem, which is meant to be called by the divider recursively when it partitions each problem. Notice that this is because the divider’s and the problem’s interpretation of the data may not be the same. When possible, this allows dividers to work on plain arrays while each particular problem reinterprets the data array as their own data structure. Of course, this may not always be possible, since non-linear data structures like graphs and trees may require more complex dividers that handle exactly the same type of data structure the problem works with.

3) solve: The solve method is in charge of converting the problem input data into its output, thus solving the base case of the problem. Once the base case problem has been solved, the combiner will take care of combining its result with the other subproblems in the same process and eventually with the results of the other processes to get the solution of the whole original problem.

E. Wrapping up

The component structure with the default classes provided by DFL is illustrated in Fig. 2. Notice that there are some default components that have not been explained for simplicity, but they allow the framework to be useful for more complex problems. For example, the DoubleAlloc is similar to the LinearAlloc but allocates twice the domain size to perform double buffering when solving time evolutions like the Jacobi problem or the heat equation. Another additional component is the BlockDivider which partitions the problem taking into account that the data is tiled in indivisible blocks like planes or cubes. Finally, the LinearCombiner combines contiguous linear boundary regions with the neighbor processes instead of just a scalar.

V. DFL D&C COMPLETE EXAMPLE

Section IV showed all the components required to implement divide and conquer algorithms with DFL, possibly redefining all of the components to customize all the parts of the algorithm. However, there is little help to the DFL programmer or DSL compiler if all the components have to be defined for each application. For this reason, DFL provides default implementations of the components for the most common use cases. Therefore, for the majority of MPI divide and conquer problems on heterogeneous architectures, only the Problem component will need to be defined.

This section will go over a simple but complete example that illustrates how a problem can be solved using all of the default divide and conquer components of DFL. The example problem we will use consists only of obtaining the sum of an array of random numbers. Although simple, this example is enough to show how all of the components are put together. Later in this document, a more complex example will be studied to justify the true value of the divide and conquer framework or DFL.

First of all, we need to define a problem that consists of summing up all the numbers of an array:

```scala
val BASE_CASE_SIZE = 32 class Sum(IN, OUT)(size: Int) extends Problem(IN, OUT) { super.allocator = new LinearAlloc(len(size)) super.divider = new LinearDivider(BASE_CASE_SIZE) super.combiner = new ScalarCombiner

override def initDomain(b: Buffer(IN), six: Int) = {
  0 until six foreach { i => b(i) = rand() }
}
```
```scala
def createSubProblem(b: Buffer[IN], siz: Int) = {
  sub = sub.clone
  sub.allocator.problemData = b
  sub.allocator.problemSize = siz
}

override def solve(b: Buffer[IN], siz: Int): OUT = {
  foreach { acc += b(i) }
  return new Solution[Float, Double](PROBLEM_SIZE)
}
```

Please note that from the DFL user level, the divide and conquer components are not used directly. Instead, the problem contains the references to them and the DFL divide and conquer runner uses them internally as we will see soon. Once the problem and all the algorithm components have been defined, we can begin creating instances and solving it from a regular DFL script:

```scala
val PROBLEM_SIZE = 4096*4096
val problem = new Sum[Float, Double](PROBLEM_SIZE)
val solution = problem.getSolution
```

The DFL class contains all the necessary information to take a problem and execute it using the divide and conquer work flow. Internally, the `solveWithDivCon` does the following:

```scala
def solveWithDivCon(IN, OUT)(p: Problem[IN, OUT]) = {
  subProbs = p.divider divide p
  foreach { sp => solveDC_recursive(sp) }
  localSol = p.combiner combine subProbs
  p.setSolution(localSol)
}
```

First, the `solveWithDivCon` function calls the recursive internal function that actually solves the problem. Then, it retrieves the solution for this process and combines it with the other processes using the specific combiner of the problem. In detail, the implementation of the `solveDC_recursive` function first checks if the problem is divisible. If the answer is positive, it splits the problem and solves each subproblem recursively. Afterwards, it combines each subproblem’s solution to obtain its own local solution. On the other hand, if the problem is not divisible (base case), the problem is solved by calling the `solve` method of the specific problem directly.

VI. Evaluation

Apart from the noticeable use of DFL as a scripting language for HPC programmers, the main design purpose of DFL is to serve as a convenient and safe target language upon which higher-level DSLs can be built with a modest effort. To use DFL for building higher-level DSLs, the languages are “stacked” in the sense that the high-level DSL translates its applications to DFL and the DFL compiler translates the application further to OmpSs. Regarding the runtime performance of DFL, preliminary results are promising and no important overheads are introduced by its abstractions, but we are still missing a thorough performance evaluation. In this section, we will review our experience and the key reasons why DFL was useful to build a complex, HPC DSL.

The HPC DSL built on top of DFL is a high-level partial differential equation (PDE) solving language for large-scale models, implemented also in Scala with LMS. Although some work has already been done in the PDE solving domain [13], we look for an additional level of expressiveness, with a declarative and safe interface to modern HPC systems. By using DFL as the target language of this DSL, the implementation can focus on just implementing the high-level abstractions of the PDE solving domain while delegating the HPC-related work to DFL.

In this DSL, the programmer first specifies a physical geometry and a set of boundary conditions on that geometry. Then, the initial state of the system is specified by means of functions. Afterwards, the equation to simulate is specified, and the DSL generates DFL and OpenCL code to automatically run the simulation on a multi-GPU architecture. In addition, some data post-processing can be specified in order to visualize the output or convert it to a scientific format for analysis tools. Skipping the most complex details, an example application of the DSL for PDEs is:

```scala
// Defining preprocess
val pre = PreProcess(source1, source2, source3)
// Defining equation
val wave = c*c * lapla(pressure) - dt2(pressure)
// Defining post-process
val post = PostProcess(snapshot each 10 steps)(VTK)
solve(pre)(post) equation wavePropagation to "wave"
```

At a very high-level, this example simulates the propagation of 3 different wavefronts on a 3D volume. The equation being simulated is the classical wave propagation equation:

\[
c^2 \nabla^2 u - \frac{\partial^2 u}{\partial t^2} = 0
\]

Where \(u\) is the pressure of the medium and \(c\) the wave propagation speed. Finally, the application specifies a post-processing step in which the state of the simulation is stored every 10 steps in VTK format. This way, after solving the equation, visual results can be obtained immediately as shown in Fig. 3, where we are visualizing a 2D slice of the volume in which the waves start hitting a compression zone at the top left (the corresponding code for the wave descriptions and the compression zone is outside the scope of this work).
When this DSL’s applications are compiled, the implementation generates both an OpenCL program that calculates one step of the simulation and a DFL application that calls it from the host after initializing the environment. The core of the generated DFL code for the example application looks as follows:

```plaintext
mainloop { i_x97 =>
  applyPointConds(pxs_x97, pyys_x97, pzs_x97, x4, x8,
    x12, pointConds_x97, x97_1, 4096, 3, i_x97)
  using Range1D(3, 1)
  PDESolverStep(0.0033330000f, x0, x4, x8,
    x97_1, x97_2, vcomps_x97, fields_x97, dconds_x97,
    coeffs_x97, 4) using ndr_x97
  expandBounds(x4, x8, x12, x97_2, periodics_x97, 4)
  using ndr_x97
  if (((i_x97) % 10 == 0) { Task(x97_2, x0)(In(x14), In(3)) {
    writeVTI(x97_2, x4, x8, x12, "wave", x0, 4,
      (i_x97)/10)
  }
}
}
```

The first kernel call in the loop (lines 2 and 3) applies the value of all three wave sources. Then, the PDE solver kernel computes a step of the equation (lines 4 and 5). After the step has been computed the boundaries of each simulation block are expanded and applied in line 6. Finally, since the application wanted a snapshot of the system each 10 steps, the last part of the loop checks the counter and signals a host-side task to print a VTI (VTK Image) file with the current state of the system.

Providing such a relatively powerful language with a simple code generator would not have been possible without the high-level support of DFL, which enables high-level DSL compilers to stay simple and focus only on domain-specific terms while still enabling them to run on complex HPC systems.

VII. RELATED WORK

HPC DSLs and its related tools are becoming popular due to the increase in architecture complexity in an effort boost domain expert productivity on supercomputing environments. This section will discuss related work on DSLs and frameworks for HPC development.

A. Domain-specific languages

DSLs, also called small languages [3], are widely used in computer science [10] and are a subject of intensive research [25]. On one hand, they are designed for a certain domain, so they are very expressive as languages for solving problems from this domain. On the other hand, due to their simplicity and size they are easy to learn and use.

DSLs can be found in many domains of computer science. To begin with, DSLs are widely used for processing text (AWK [1], for instance) and to generate parsers, like [8]. Other DSL examples include VHDL and Verilog for electronic circuit design and Matlab [14] for numerical computing.

With advent of multicore architectures, many emerging programming languages implement parallelism and concurrency as one of their paradigms, and DSLs are no difference. One of such languages is OptiML [23], a parallel DSL for implementing machine learning algorithms. Another example of HPC DSL is Liszt [6], a language for building parallel mesh-based PDE solvers. In particular, Liszt was implemented using Scala-Virtualized like DFL, but instead of using LMS it was implemented via a compiler plugin. The Liszt plugin for the Scala-Virtualized compiler type-checks the code and emits a specific intermediate representation that is further optimized and translated into C++ or CUDA C.

B. Building domain-specific languages

This subsection focuses on methods that allow DSL developers to embed DSLs in general-purpose languages. One of the first languages that introduced a means to implementing embedded DSLs is Lisp [22]. Lisp offers powerful macro systems that allows programmers to extend the syntax of the original language. Then, macros are expanded at preprocessing time into regular Lisp code, which makes Lisp macros a good fit for DSLs. Finally, it is worth noting that the design of Scala Macros is inspired by the Lisp macro system in an effort to bring compile-time metaprogramming to the already extensive Scala language.

Lisp macros are an approach to static metaprogramming and code generation, and a similar idea was implemented in Template Haskell [20]. Haskell does not have a macro-like system that is part of the language. However, the same compile-time metaprogramming was achieved by several extensions in the language, such as quasi-quotations. This feature implements a set of functionalities such as fresh name generation or error reporting, needed for compile-time code generation.

So far, we have only referred to solutions targeting compile-time code generation. One of the platforms that allow runtime code generation is MetaOCaml [24], [17]. This language is a set of extensions of the original OCaml language to enable dynamic code generation. In addition, generated code is guaranteed to be type-safe at compile time. As a special remark, the language is based on the concept of multi-stage programming [24], much like LMS (see Sec. II-B). The idea is that some computation can be divided into stages, depending on information availability. In MetaOCaml, the programmer is equipped with language constructs that allow explicitly specifying which parts of applications should be staged (compiled).

Regarding HPC DSL production, one of the most mature approaches is the Delite Framework [4], a library built on top of LMS to support the implementation of high performance
DSLs by defining DSL operations in terms of parallel constructs. In other words, it is like an extension of LMS to enable HPC DSLs to be embedded in Scala. The Delite framework comes with its own runtime system and generates both the host applications and the accelerator OpenCL/CUDA code. Such an approach is robust and interesting, as a safe interaction is guaranteed, while leaving room for useful aggressive optimizations. However, the DSL implementor can only use the Delite API to define the accelerator code, which results in some restrictions in terms of which operations can be implemented. Moreover, having no support for already written libraries or algorithms can be a problem when trying to integrate DSL implementations with large-scale HPC systems and libraries.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented DFL, a DSL designed to exploit distributed and heterogeneous HPC systems. The goal of DFL is to serve as the target language for other DSLs, providing a convenient and compiler-friendly interface that simplifies the development of DSLs for HPC environments. DFL leverages the LMS framework for the DSL compiler infrastructure and the hybrid MPI/OmpSs programming model for the DSL runtime system. This paper describes the main features of DFL and how they have been successfully used to implement a DSL to solve partial differential equations on top of it. Additionally, DFL can also be used by HPC programming experts directly as a user-level language for developing/prototyping HPC applications, leveraging its features to program heterogeneous HPC systems.

Although DFL is already fully functional for single node environments, we are currently working to support more high-level operations to widen DFL’s usability and usefulness on distributed environments instead of just MPI generation for divide and conquer. The additional high-level operations we are considering include map, reduce, master-slave and pipelining. Moreover, future work will also extend DFL to leverage advanced OmpSs features such as reductions [5].

ACKNOWLEDGMENT

This work has been developed with the support of the grant SEV-2011-00067 of the Severo Ochoa Program, awarded by the Spanish Government, by the Spanish Ministry of Science and Innovation (contracts TIN2012-34557, and CAC2007-00052) and by the Generalitat de Catalunya (contract 2014-SGR-1051).

REFERENCES