Designing the Language Liszt for Building Portable Mesh-based PDE Solvers

Zachary DeVito
Stanford University
E-mail: `zdevito@stanford.edu`

Pat Hanrahan
Stanford University
E-mail: `hanrahan@cs.stanford.edu`

Abstract. Complex physical simulations have driven the need for exascale computing, but reaching exascale will require more power-efficient supercomputers. Heterogeneous hardware offers one way to increase efficiency, but is difficult to program and lacks a unifying programming model. Abstracting problems at the level of the domain rather than hardware offers an alternative approach. In this paper we describe the design of Liszt, a domain-specific language (DSL) for solving partial differential equations on meshes. Many domain-specific languages and frameworks have been proposed for physical simulation. Liszt is unique in that it targets current and future heterogeneous platforms. We have found that designing a DSL requires a careful balance between features that allow for automatic parallelization and features that make the language flexible.

1. Introduction

The desire for increasingly accurate and complex physical simulation has driven the need for more powerful supercomputers. Until recently, these computers were typically composed of a cluster of homogeneous machines connected with a high-speed interconnect. The desire to reach exascale computing will require more power-efficient designs.

One way to increase efficiency is to use accelerators. By specializing the hardware, it is possible to build hardware that executes more FLOPs per watt. Recent supercomputers such as the Tianhe-1A, or Cray's XK-6 contain GPU accelerators to improve efficiency [22, 9]. Complicating things further is the heterogenoeus nature of these platforms; they contain both CPU and GPU nodes. Heterogeneity is necessary because applications contain a mix of sequential and data-parallel code. The problem is that heterogeneous systems are more difficult to program; a programmer needs to use multiple programming models such as MPI, pthreads, and CUDA/OpenCL [21, 19] to implement a single application.

One approach to solving the problem of programming complex hardware is to raise the level of abstraction. The idea is to trade-off generality for domain-specificity. In the same way that hardware has gained efficiency by specialization, specialized software can also perform more efficiently. Raising the level of abstraction also has the advantage that it leads to a more productive programming environment. High-level domain-specific languages (DSLs) such as Matlab and R are already widely used in computational science because they speed rapid prototyping and experimentation.

The domain we tackle is physical simulation, in particular, the case of solving partial differential equations (PDEs) on meshes. Several domain-specific languages already exist to help solve these equations. One popular platform is OpenFOAM. In OpenFOAM a programmer expresses the problem in code as a PDE [29]. Figure 1 shows an example PDE and its equivalent representation as expressed in OpenFOAM.

This domain-specific approach has several advantages. Since the program does not specify a machine-specific implementation, a developer of OpenFOAM is free to create an implementation of the PDE solver for any architecture. This design allows a single problem specification and provides the potential for forward portability to future heterogeneous architectures. Furthermore, a higher-level abstraction typically makes programming these applications more productive since low-level details of the parallel implementation are handled automatically.

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot \phi \mathbf{U} - \nabla \cdot \mu \nabla \mathbf{U} = -\nabla p$$

```
solve
    (
        fvm::ddt(rho, U)
      + fvm::div(phi, U)
      - fvm::laplacian(mu, U)
        ==
      - fvc::grad(p)
    );
```

Figure 1: A partial differential equation and its equivalent OpenFOAM representation as described in the OpenFOAM user guide [23].

The portability and productivity of domain-specific design come at the cost of flexibility in expression. OpenFOAM provides a set of discretization approaches to solve PDEs, but researchers often need to develop their own numerical techniques, requiring them to extend the library itself [29].

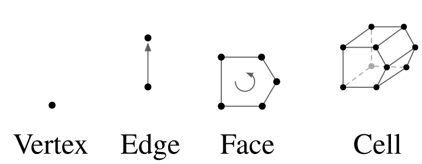
Many challenges arise in a designing a DSL. First, a DSL should make it possible to express the most important problems in the domain. Second, it should be clean and elegant so that programmers are productive. Third, it should be possible to implement efficiently on a range of parallel hardware. With these challenges in mind, we have developed a new domain-specific language for solving PDEs on meshes called Liszt. The same Liszt code can be compiled to run on clusters, traditional SMPs, and GPUs; it performs comparably to hand-written code on each architecture.

While designing Liszt, we have found that it is not sufficient to simply add high-level abstractions of the domain to a general-purpose language. A DSL must carefully consider the performance implications of each language feature, in some cases restricting the language to ensure that the code will run efficiently. In particular, our approach to automatically parallelizing code on different architectures relies on the ability of our platform to determine the data dependencies of any Liszt expression. To do this, we tie all data accesses to the mesh of the decomposed domain.

In this paper, we discuss the design decisions that we made while developing Liszt. Furthermore, we present different possible designs and compare our choices to those made in related systems.

2. Language

To make the discussion of design decisions concrete, we start with a simple example of a complete Liszt application that models heat conduction; we follow with a discussion of the design. Figure 2 shows Liszt code for calculating equilibrium temperature distribution on an unstructured grid of rods using Jacobi iteration. Unlike the OpenFOAM example, Liszt code is written at the level of the discretized mesh representing the physical domain rather than at the level of the continuous PDEs. The mesh is expressed in terms of abstract data types:



```
1  //Initialize data storage
2  val Position =
3      FieldWithLabel[Vertex,Float3]("position")
4  val Temperature = FieldWithConst[Vertex,Float](0.f)
5  val Flux = FieldWithConst[Vertex,Float](0.f)
6  val JacobiStep = FieldWithConst[Vertex,Float](0.f)
7  //Set initial conditions
8  val Kq = 0.20f
9  for (v <- vertices(mesh)) {
10    if (ID(v) == 1)
11       Temperature(v) = 1000.0f
12    else
13       Temperature(v) = 0.0f
14 }
15 //Perform Jacobi iterative solve
16 var i = 0;
17 while (i < 1000) {
18     for (e <- edges(mesh)) {
19        val v1 = head(e)
20        val v2 = tail(e)
21        val dP = Position(v2) - Position(v1)
22        val dT = Temperature(v2) - Temperature(v1)
23        val step = 1.0f/(length(dP))
24        Flux(v1) += dT*step
25        Flux(v2) -= dT*step
26        JacobiStep(v1) += step
27        JacobiStep(v2) += step
28     }
29     for (p <- vertices(mesh)) {
30        Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
31     }
32     for (p <- vertices(mesh)) {
33         Flux(p) = 0.f; JacobiStep(p) = 0.f;
34     }
35     i += 1
36 }
```

Figure 2: Example code written in Liszt that calculates the temperature equilibrium due to heat conduction.

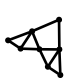
Liszt supports unstructured meshes of arbitrary polyhedra in three dimensions. It handles the details of loading the mesh; an application programmer interacts with it using built-in operators. For simplicity, we currently do not allow the topology of the mesh to change during program execution.

In our example code, lines 2–6 initialize a number of *fields*. A *field* is a parameterized type that implements an associative map from a mesh element to value similar in concept to the fields in Sandia's Sierra framework [28]. Fields are designed to hold the degrees of freedom that are used to approximate a continuous function defined over the domain using a set of basis functions.

Lines 9–14 use a parallel *for-comprehension* to set up an initial condition with temperature concentrated at one point. A *for-comprehension* expresses computation for all elements in a set. We use the term comprehension rather than loop since it does not contain any loop-carried dependencies. The *for-comprehension* operates over a topological set, `edges(mesh)`, that contains all edges in the discretized mesh. Though not shown in the example, nested *for-comprehension* are supported.

Lines 17–36 perform 1,000 Jacobi iterations to solve for the steady-state temperature distribution. During each iteration, the expressions `head(e)` and `tail(e)` extract the mesh vertices on either side of the edge. These topological functions relate mesh elements to neighboring elements. Figure 3 summarizes the topological functions built into Liszt. The head and tail are then used as keys to extract data from

```
BoundarySet¹[ME <: MeshElement](name : String) : Set[ME]
```



```
vertices(e : Mesh) : Set[Vertex]
cells(e : Mesh) : Set[Cell]
edges(e : Mesh) : Set[Edge]
faces(e : Mesh) : Set[Face]
```

[1] A subset of elements; defined in a configuration file

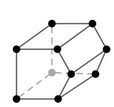
[2] The faces/edges are oriented counter-clockwise (CCW) around element *e*

[3] The dual operators of head and tail, returning the cells on either side of the face

[4] Flips the orientation of the element (e.g. **head**(e)=**tail**(**flip**(e)))

[5] Orients the element *e* to point towards *d* (e.g. **head**(**towards**(e,v)) = v)

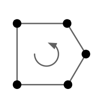

```
vertices(e : Vertex) : Set[Vertex]
cells(e : Vertex) : Set[Cell]
edges(e : Vertex) : Set[Edge]
faces(e : Vertex) : Set[Face]
```



```
cells(e : Cell) : Set[Cell]
vertices(e : Cell) : Set[Vertex]
faces(e : Cell) : Set[Face]
edges(e : Cell) : Set[Edge]
```



```
vertices(e : Edge) : Set[Vertex]
facesCCW²(e : Edge) : Set[Face]
cells(e : Edge) : Set[Cell]
head(e : Edge) : Vertex
tail(e : Edge) : Vertex
flip⁴(e : Edge) : Edge
towards⁵(e : Edge, t : Vertex) : Edge
```



```
cells(e : Face) : Set[Cell]
edgesCCW²(e : Face) : Set[Edge]
vertices(e : Face) : Set[Vertex]
inside³(e : Face) : Cell
outside³(e : Face) : Cell
flip⁴(e : Face) : Face
towards⁵(e : Face,t : Cell) : Face
```

Figure 3: Liszt's major built-in topological relations. For every operation on a vertex (edge) there is a dual operation for the cell (face).

fields.

The *Reduction* operators on lines 24–27 are used to calculate new values for the `Flux` and `JacobiStep` at each iteration. Since *for-comprehension* is a parallel operator, the semantics of the language dictate that these reductions occur atomically. In addition to summations, other associative and commutative operators such as multiplication or min/max are provided.

3. Design

The design of a language or library for solving PDEs must balance competing goals of flexibility, productivity, and efficiency. Here we examine several of the important decisions that were made while designing the Liszt language. First, we chose to abstract the domain at the level of the elementwise formulation. At this level, the problem is expressed in terms of basis functions defined over discrete elements. An elementwise formulation must present an abstraction for interacting with the discretized domain and storing data for each element to represent fields. Liszt chooses the approach of using a discretized mesh and topological functions to represent the domain. Furthermore, the compiler or framework must transform code based on the chosen abstraction to an efficient parallel implementation for a given parallel programming model. To implement this abstraction in parallel, Liszt automatically infers the data dependencies in each operation using program analysis and provides language semantics that promote parallel code. We provide a simple interface for using external libraries. We examine the rationale for each of these decisions in detail.

3.1. Choosing the level of abstraction

When developing methods to solve PDEs, scientists reason at several levels of abstraction. These levels provide a natural starting point for exploring the space of possible designs. We examine three separate levels here that have been used to create successful libraries for solving PDEs.

Continuous PDEs The highest level defines a set of governing equations over a spatial domain. In OpenFOAM [29] and Sandia's Sundance [15], the programmer represents the problem in terms of the PDEs themselves using abstract data-types representing fields and operators. The FreeFEM and FEniCS projects also use similar formulations [25, 13]. The particular numeric discretization and solvers are provided by the library and are configurable in the API. These libraries then also provide parallel implementations for solving the discretized problem. OpenFOAM, for example, provides an MPI-accelerated implementation [23].

Elementwise Formulation In order to solve the continuous problem formulation, a second level of abstraction is introduced where the domain is discretized into a mesh of elements; continuous fields are approximated with a set of basis functions with local support in each element. Liszt itself uses the elementwise formulation as its starting point: computation and data are expressed in terms of the discretized mesh; the specific data-structures or model for parallelization are left abstract. In addition to Liszt, various frameworks take this approach. Giles et al.'s OP2 library expresses the mesh as a set of user-defined relationships between elements [14]. SBLOCK, a solver for structured meshes [5], expresses computation as kernels that operate on a local neighborhood around an element. Sandia's Sierra framework also expresses computation in terms of the discretized mesh but incorporates communication primitives that assume parallelization is based on domain partitioning [28].

Data structures and parallel programming At the lowest level, the programmer creates specific data structures for storing the mesh and discrete data needed to represent the solution for a parallel programming model. Many production PDE solvers are written this way, typically using MPI as the underlying programming model. Libraries that support this style of programming, such as PETSc or Sandia's Sierra framework, provide utility functions for partitioning the mesh across different memory spaces and communicating between them [4, 28]. The framework used in Stanford's

PSAAP program follows this approach [24]. The framework partitions the mesh across different MPI nodes. On each node, code is written in a single-threaded style expressed as computation on the local partition. When nonlocal data is required, specific calls are inserted to perform communication via MPI.

Continuous PDEs are ideal for users who have a particular problem they want to solve using already established numeric methods. Since the problem is represented mathematically, the library itself can provide different implementations that are efficient for different architectures. However, this level of abstraction is less useful for researchers who want to experiment with new numerical methods that are not already built into the library. While both OpenFOAM and Sundance can be extended with additional models, certain extensions require the user to be aware of the underlying parallel implementation [29, 15] and may require different implementations to run on new architectures such as GPUs.

The expression of the problem at the level of data structures in a parallel programming model gives the programmer the most flexibility in choice of numeric method and solvers. Since this lower level of abstraction exposes the features of the underlying parallel programming model, it may not run efficiently on architectures that do not match the programming model. For example, frameworks such as Sierra and Stanford's PSAAP solvers use an abstraction based on mesh partitioning [28, 24]. On modern graphics cards that have thousands of threads of execution, each partition will be very small, resulting in a lot of communication between partitions.

We chose to abstract at the level of the elementwise formulation because it provides the flexibility of the low-level approach to express different numeric formulations while keeping the details of the parallel implementation abstract. This abstraction provides the potential for automatic portability to different parallel programming models.

3.2. Abstracting meshes and fields

The elementwise formulation is based on a discretized domain. A language or library for the elementwise formulation must provide a way to access the elements of the domain and store data for each element. Several approaches are commonly taken in practice.

Ad hoc Storage A common approach to representing the domain is with ad hoc data structures. For instance, in Stanford's PSAAP solvers, the connectivity of the mesh needed by the application is provided in arrays using the compressed row storage (CRS) format [24]. Elements themselves are assigned contiguous integer IDs, and data is stored in arrays addressed by ID. Typically the frameworks build only the connectivity that is needed for the particular applications being handled, leading to representations that are inconsistent across different programs.

Mesh Topology One way to represent this domain in a consistent fashion is with a mesh of topological elements. Liszt takes this approach by providing a three-dimensional mesh representing a three-dimensional manifold. Neighboring mesh elements are extracted using topological functions. To ensure we have a complete set of topological functions suitable to a wide range of programs, we designed our operators based on the primitives suggested by Laszlo and Dobkin [11]. Figure 3 summarizes their behavior.

Relations An alternative method, taken by OP2, is to create a set of user-defined relations over elements [14]. These elements may represent mesh topology such as vertices, but they are user-defined, so they conceptually can represent other entities such as the nodes of a higher-order finite element. Relations define a 1 to N mapping from one type of user-defined element e to another type e':

$$r(e') = \{e'_0, e'_1, ..., e'_N\}.$$

For instance, a relation may represent a 1-to-2 mapping of edges to their adjacent vertices:

$$\text{edges}(e_0) = \{v_0, v_1\}$$

These relations are defined explicitly by using a graph data-structure that maps elements from the domain to the set of elements in the range.

Each of these designs has different advantages. Ad hoc storage based on arrays and integer indexing is flexible and can be tailored to a specific problem, but the lack of a common interface makes it difficult to incorporate into platforms that need to generate parallel code on different architectures. Relations are also flexible, but they still require the user to define an appropriate set of connectivity for the program by hand.

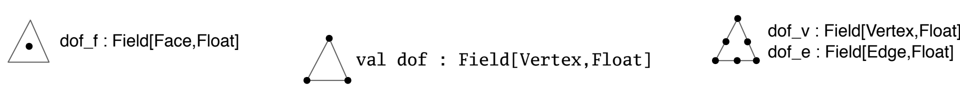
We chose to use mesh-based topological functions since they are an intuitive and familiar way of working with an element and are closely related to the way programmers visualize the domain. Additionally, using the mesh as the basic data structure provides additional semantic knowledge of mesh relationships, which can be used by the platform to produce more efficient algorithms. For instance, Liszt can use the facet-edge representation suggested by Laszlo and Dobkin [11] to create efficient data structures for partitioning a mesh across many cluster nodes. This is especially useful for large problems, where the mesh cannot fit in the memory of a single machine, since writing code that can load the mesh in parallel for ad hoc representations is tedious.

Building the semantics of the mesh into the language also offers additional opportunities for the specialization of data-structure for particular problems. For example, if the input mesh is a regular grid, it would be possible to implement the topological functions using simple affine transforms and provide better implementations of operations such as mesh partitioning.

In addition to representing a domain, an abstraction must provide a way to store user-provided data. For ad hoc data structures based on integer indexing, programmers usually store data in arrays indexed by element ID. Relational models such as OP2 let the programmer store data in associative maps from user-defined elements to values [14].

Liszt's mesh-based representation naturally leads to storing data at mesh elements using fields. For this storage we specifically decided to use a struct of arrays form (e.g., `Temperature(cell)`), rather than an array of structs form (e.g., `cell.Temperature`). We believe that the struct of arrays form, where fields are defined independently of the element, is more intuitive when a single program is solving multiple equations each with a different set of unrelated fields.

Mesh-based storage works well for simple programs. For instance, piecewise linear basis functions for triangular elements can be represented by using a field defined over faces. For continuity across the domain, linear basis functions can be represented using three degrees of freedom, one at each vertex, coupled to neighboring elements using a field defined at vertices. However, storing the degrees of freedom for a quadratic triangular element requires six degrees of freedom, three at each vertex and three along the edges of the triangle. In Liszt we can represent this with two fields, one on the edges and one on the vertices:



Constant Linear Quadratic

However, using multiple fields in this way in awkward and becomes increasingly complicated for higher-order elements. In this case, a user-defined relation mapping an element directly to its degrees of freedom would be more intuitive. Despite this complexity, we chose to use element-based storage to make the most common operations simpler to implement. As we continue to develop Liszt, we may consider making these our built-in functions extendable to address the ability to work with higher-order elements.

3.3. Inferring data dependencies with stencils

An implementation of a DSL must be able to translate code that uses abstractions such as meshes and fields into an efficient program. In order to produce efficient code on modern architectures, a domain-specific compiler must be able to find parallelism, expose memory locality, and minimize the synchronization of parallel tasks. Since the same operations are mapped to each element, the problems have a large degree of data parallelism. Since the basis functions are typically designed to have local support, computation will access data only from local topological neighborhood around the element. We refer to this neighborhood as the computation's *stencil*. Furthermore, since computation occurs in data-parallel phases, synchronization is typically required only between phases.

To exploit these features of the domain, a DSL must be able to extract this information from the program. One approach is to reason about the data dependencies for a computation. If the DSL can determine the set of reads and writes that an arbitrary expression will perform, it can determine whether two expressions can be run in parallel, schedule operations using the same data to run locally (either spatially or temporally), and insert synchronization where it is necessary to fulfill a dependency. Since PDE solvers are typically iterative, the DSL can perform this reasoning before the first iteration and reuse it for subsequent iterations.

Determining data dependencies is difficult in general since it requires reasoning about the behavior of arbitrary programs. As an example, the statement `A[`$f_1(i)$`] = B[`$f_2(i)$`]` requires the DSL to understand the behavior of $f_1(i)$ and $f_2(i)$, arbitrary expressions. Libraries for solving mesh-based PDEs solve this problem by restricting f in different ways.

Affine partitioning For the restricted set of problems that considers only regular grids, the data needed in a single element is almost always an affine transformation of the indices of the element. In this case affine partitioning techniques can automatically determine efficient parallel implementations when $f(i)$ is an affine expression [20]. The PIPS compiler, used by many scientific programs, takes this approach to parallelize code [2].

Explicit Provided Dependencies Alternatively, the dependencies can be provided explicitly. In the case of regular grids, the halo of data required for an element, e_{ij}, can be declared explicitly for each element as mathematical functions of i and j; this is the approach taken by SBLOCK [5].

Dependencies can also be explicitly defined for unstructured problems. In OP2, parallel computation is then expressed as kernels that can gather and scatter data only from a particular user-defined relation [14]. Relations are passed explicitly to the kernel, so the data used at element e are precisely the data store on e and any data stored on elements in $r(e)$ for the relationship r.

Inferring Dependencies with Program Analysis Another approach is to infer the dependencies from the use of built-in operators. Liszt can automatically determine data dependencies through program analysis of its build-in mesh functions and the way that fields are used. To accomplish this inference, Liszt makes the following semantic restrictions that enable our compiler to perform program analysis to discover dependencies automatically:

- The mesh topology does not change.
- Mesh elements can be obtained only through topological functions, and data in fields can be accessed only by using the mesh element as the key.
- Assignments of variables to topological elements and fields are constant (i.e., there is a single static assignment of a value to any variables that refers to mesh elements). This restriction ensures that a while-loop cannot traverse the mesh.

Section 4.1 gives an overview of the approach to infer the dependencies.

Each approach has different trade-offs. Affine partitioning typically does not work for unstructured meshes, where indices into arrays have data dependencies on the mesh. Explicitly provided dependencies can work for unstructured meshes, but it can be difficult to express more complicated dependencies. For instance, in OP2, nested use of the relations is not allowed since it could result in dependencies for element e that are not contained in $r(e)$ [14]. For more complicated dependencies, the programmer must explicitly create a new relation to represent them.

We chose to automatically infer the dependencies since doing so enables the programmer to code in a style that is similar to single-threaded code but still produce a parallel implementation for unstructured meshes. While performing this inference requires some restrictions on the language, we have found them acceptable in practice since they enable the application programmer to express code without worrying about explicitly declaring the dependencies. This approach allows nested operations to be expressed naturally as nested *for-comprehension*.

3.4. Choosing language semantics for parallelism

While analyzing data dependencies is useful for determining a parallel implementation, it is also helpful to add additional language features to encourage the programmer to write code that is parallelizable. Liszt adds three additional language features to express the program in a parallel way: *for-comprehension* are parallel operators, *reduction* operations like `+=` occur atomically, and *fields* have *phases*—during a *for-comprehension* a field must be read-only, write-only, or reduce-only.

Consider an alternative design in which *for-comprehension* are implemented as loops and *reduction* operators are not atomic. Given this design, the reductions to `Temperature` and `JacobiStep` in Figure 2 would introduce a loop-carried dependency across the loop, forcing the writes to `Temperature` and `JacobiStep` to serialize. These flux calculations are common. By providing parallel *for-comprehensions* and commutative/associative *reductions*, Liszt is free to reorder these writes, allowing for efficient parallel implementations.

Field *phases* were added to reflect the fact that data-parallel operation in a PDE solver typically produces data only for later operations. The phasing of fields in Liszt is handled automatically by the compiler. This restriction ensures that a parallel *for-comprehension* does not introduce any nondeterministic values for a given implementation. OP2 takes a similar approach to reduce dependencies in data-parallel operations by having the user explicitly annotate data storage as read only, write only, or sum only during a kernel [14].

3.5. Working with external libraries

Since certain operations will fall outside the expressibility of a domain-specific language, a way to interact with external libraries is necessary. Liszt currently has rudimentary support for calling external libraries, with more support for common libraries such as sparse matrix solvers left as future work. For calls to external libraries, Liszt allows the declaration of external functions.

```
def foo(a : Int, b : Float) = __
```

The user must then provide an implementation of `foo` for every runtime that Liszt targets (e.g., a C implementation for our MPI runtime and a CUDA implementation for our GPU runtime). This makes it easy to call routines written as libraries in other languages.

External interactions with data structures such as the mesh or fields are more complicated since Liszt must reason about the data dependencies of code that uses these structures. Interfacing with sparse matrix solvers is an especially common case of this problem. We plan to address these interactions by creating a way to represent sparse matrices inside of Liszt and then making it easy to combine this representation with external solvers. This design will allow the programmers to choose the solver most suited to their problem but still write the code that assembles the matrix in Liszt. An important choice for this design is how to represent the rows and columns of the matrix in a way that interoperates well with the mesh elements and fields that Liszt provides. Discussion of this issue is beyond the scope of this paper, but one possible way to accomplish this is to address the rows and columns of the matrix with mesh elements.

4. Implementation

The design of Liszt balances the concerns of flexibility, productivity, and efficiency by providing constructs that operate at the level of the elementwise formulation, while still allowing the compiler

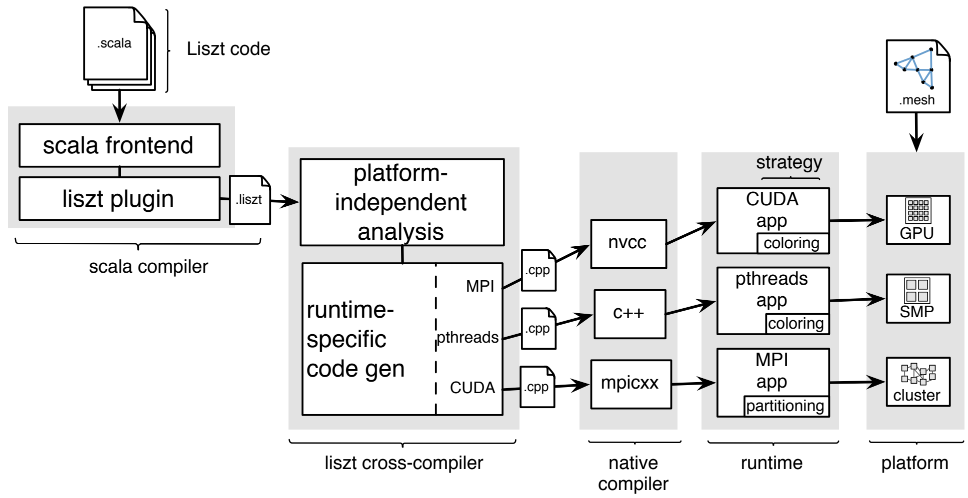


Figure 4: Overview of the stages of our Liszt implementation. Code is preprocessed by a plugin in the Scala compiler to produce an intermediate representation. The Liszt cross-compiler then produces code for one of our three runtimes.

to automatically determine the data dependencies. To show that this design effectively captures the necessary information for efficient implementations, we have created a compiler for Liszt that can produce code for three parallel platforms: an SMP, a large cluster, and a modern GPU. To target these architectures, we use three runtimes: one based on MPI, one on pthreads, and one on CUDA. This approach demonstrates our ability to run across different programming models. Furthermore, we demonstrate two different approaches to parallelizing Liszt code. The first uses a partitioning-based approach to target programming models such as MPI that have disjoint memory spaces. The second, used in our shared-memory runtimes (pthreads, CUDA), determines how to parallelize a *for-comprehension* using a graph-coloring approach.

We present an overview of the implementation here. DeVito et al. provide a more detailed description [10]. Figure 4 gives an overview of the architecture. Liszt's syntax is a subset of the Scala programming language. A compiler plugin in the Scala compiler generates an intermediate representation for Liszt code that is passed into a separate Liszt-specific cross-compiler. We chose to use Scala as our front-end since it provides constructs that are suited to creating embedded domain-specific languages [26]. Its rich type system allows expression of Liszt's abstract data types such as mesh elements and files, while also providing type inference to reduce the need for explicit type annotations in code. Additionally, its support for abstract *for-comprehensions* made it simple to express Liszt programs without the need to create new syntax. The language features of Scala also allow domain-specific languages to be embedded directly in the language as a library. Chafi et al. present an overview of this approach—including an implementation of Liszt—that can produce an intermediate representation of DSL code suitable for performing domain-specific optimizations at runtime [6].

After generating the intermediate representation from Scala code, the Liszt cross-compiler performs some runtime-independent program transformations. In particular, it generates code that when given a particular mesh can produce the data dependencies for the Liszt program. This *stencil* code is then used during the initialization of a Liszt application in the implementation of our partitioning or coloring approaches. The cross-compiler additionally inserts explicit `enterPhase` statements into the code where fields change phase using a data-flow analysis. Runtimes can use these statements to perform actions such as cluster communication when a field changes from a write phase to a read phase.

A second stage of the cross-compiler then generates C++ or CUDA code for our specific runtimes,

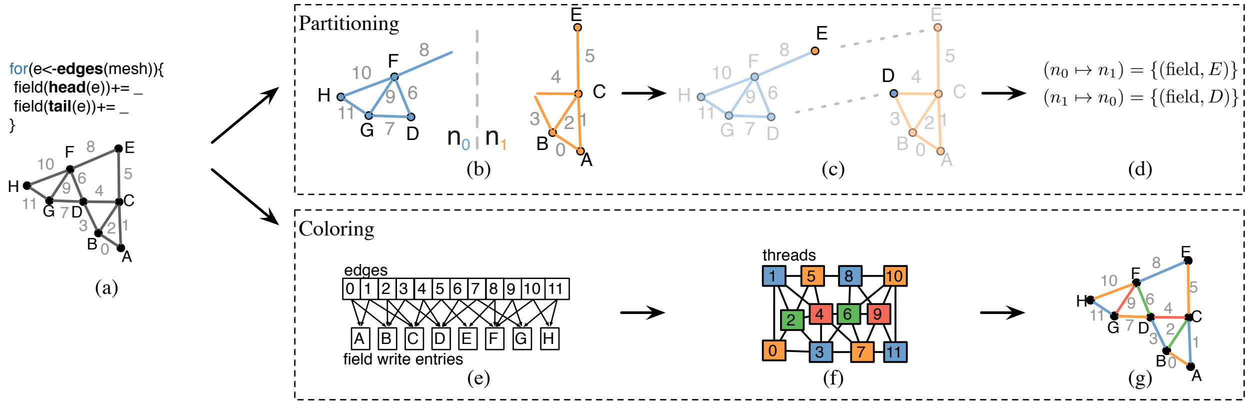


Figure 5: For a particular input (a), Liszt can parallelize the code using either a partitioning or coloring strategy. When using partitioning, Liszt first partitions the mesh to different nodes (b); it then uses the stencil to discover ghost elements (c), and establish communication patterns to update ghost elements (d). When using a coloring approach, Liszt builds a map from elements in a *for-comprehension* to the field locations that the element will read (e). This graph is used to build an interference graph between elements (f). Coloring the interference graph results in a schedule (g) where elements of a single color can run in parallel.

which is handed off to a general-purpose compiler to create the final executable.

4.1. Stencil

Regardless of the runtime, each application relies on the ability of Liszt to determine the data dependencies inside a *for-comprehension* given a particular mesh. In particular we define a stencil:

$$\mathcal{S}(e_l, E) = (R, W)$$

where e_l is a Liszt expression, E is an environment mapping variables to values, and R and W are the of reads and writes that e_l may perform. Our approach to implement this stencil works in two stages. We first transform a given Liszt expression e_l into a new expression e_l' such that e_l' is guaranteed to conservatively approximate the data dependencies of e_l—that is, e_l' reads and writes a superset of the values of e_l. Additionally, we enforce that e_l' must terminate. This first stage can be done in the cross-compiler since it does not rely on the topology of a particular mesh. The second stage runs at program initialization. To evaluate $\mathcal{S}(e_l, E)$ for an expression e_l, we execute e_l' in environment E and record the reads and writes it makes as the set of pairs (f, e), where f is the field being accessed and e is the element used to access it.

To transform from e_l to e_l' is conceptually simple. *If*-statements are transformed such that both paths are executed, ensuring the new expression e_l' can only read or write more entries than e_l. *While*-loops are transformed such that they execute their body exactly once. The design of Liszt ensures that variables referring to mesh elements cannot be reassigned, so executing the body of the loop a single time captures all the accesses that the loop can perform. The resulting expression must terminate since mesh sets are of constant size and Liszt does not contain any looping constructs besides *while*-loops (in particular, recursion is not supported).

4.2. Implementing the execution strategies

Given the *stencil* in the form of the transformed expression e_l', we can parallelize code using different strategies depending on architecture. Each of our three runtimes parallelizes Liszt code using either a partitioning strategy (MPI) or coloring-based strategy (pthreads, CUDA). Here we briefly describe each strategy to illustrate how the stencil is used. Figure 5 summarizes the stages performed in each approach.

4.2.1. Partitioning Our partitioning strategy is based on a common design-pattern for implementing PDE solvers using MPI. However, unlike most frameworks that require users to explicitly insert communication statements, Liszt handles all the details of internode communication automatically. Figure 5 gives an overview of this strategy. First, for a particular mesh and program (a), the mesh is loaded and partitioned across N nodes using an external partitioner (b). We have found that using ParMETIS on a graph of cell-cell connectivity works will for most three-dimensional problems [18]. When an unnested *for-comprehension* executes, each node will perform computation for elements of the set that are in its partition.

Since computation for element e will access data in a neighborhood around e, elements near the border of a partition may need to access data on a remote partition. In order to make this access efficient, that data is duplicated locally and stored in *ghost* elements. The partitioning approach discovers these ghost elements at initialization using the stencil on its local partition to calculate the sets of reads and writes that a partition will perform. Accesses found by the stencil that are not local determine the presence of ghosts. For example, in Figure 5 (c) vertex E is discovered to be a ghost on node 0.

If during a write- or reduce-only *phase* a node n_0 writes data to an element that is a ghost on some node n_1, then n_0 must send n_1 the updated value (or delta, in the case of reductions). Since the field cannot be read until the field enters a read-only *phase*, this communication needs to be performed only inside the `enterPhase` statements added by the compiler when the field enters a read-only phase. This approach allows the runtime to batch communication and delay synchronization until the values are needed remotely. For additional efficiency we can use the stencil to precompute this pattern of communication, as illustrated in Figure 5 (d).

4.2.2. Coloring In a shared-memory system, we can avoid duplicating data and creating ghosts by using a scheduling approach based on graph coloring rather than partitioning [1, 16]. This alternative can be beneficial on runtimes where there are many parallel threads of execution. In this case, the partitions would be small, resulting a large amount of duplication and communication along the boundaries. Liszt targets this approach automatically.

Figure 5 illustrates the process. To parallelize a *for-comprehension*, we group mesh elements in the set into batches such that no two elements in the same batches write to the same field entry. Each batch is run in parallel separated by global barriers. We can use the stencil to determine the writes for each element and create a bipartite graph connecting elements to the field entries they will write (Figure 5 (e)). We use this graph to create a new graph between mesh elements where an edge exists if they write the same field entry (Figure 5 (f)). Coloring this graph using the algorithm of Chaitin et al. groups the elements into batches of independent work [7].

In our GPU runtime, each unnested *for-comprehension* is transformed into a CUDA kernel and invoked once per batch. Our pthreads runtime operates similarly, with multiple-threads dividing the batches using work-stealing, and each batch separated by a global barrier.

5. Performance

We evaluate the performance of our compiler and runtimes using four example applications to demonstrate that our design is capable of taking a single program and running it efficiently across different architectures and applications. We compare this performance to hand-written implementations

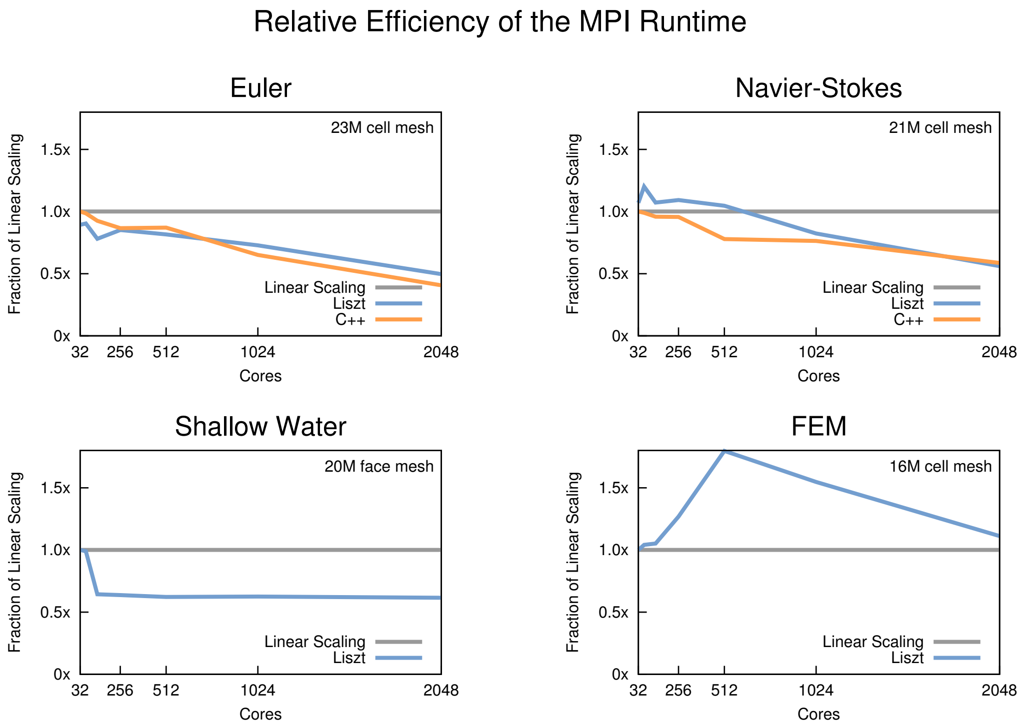


Figure 6: The relative efficiency Liszt applications on a 2048-core cluster as compared to ideal linear scaling of the 32-core run. The Euler and Navier-Stokes applications are compared against their hand-written MPI implementations. The Shallow Water and FEM applications are compared against their own 32-core performance, since they were ported from nonparallel code. Each node of the cluster contains two Intel Nehalem X5650 processors running OpenMPI 1.4.2. We measured performance using 4 through 256 nodes, using 4 cores per processor. The FEM code shows superlinear scaling from 256 to 512 cores as the partitions become small enough to fit into L3 cache.

when available or to the published performance of similar implementations. A more detailed analysis of the performance of our implementation is available in DeVito et al [10].

Our main test applications are ports of applications used in Stanford University's DOE PSAAP Project. This project is simulating the unstart phenomenon in hypersonic vehicles [24]. We ported the core Euler and Navier-Stokes (NS) fluid dynamics solvers. Both applications are cell-centered, using face-based flux calculations and a forward Euler time-step. The Euler solver calculates inviscid flow, while the NS solver additionally calculates viscous flow, allowing us to test different sized working sets. Additionally, we implemented a shallow water simulator (SW) based on an algorithm by Drake et al. [12] and a simple finite-element method (FEM) that computes the Laplace equation on a cubic grid. All applications use double precision numbers. These applications provide us with a variety of working set sizes and arithmetic intensities.

To measure the overhead of using Liszt, we compared each application to a hand-written C++ counter-part. The Euler and Navier-Stokes solvers have reference implementations written in MPI; for these applications, we also compared Liszt's MPI runtime to the reference implementations on both single SMPs and a large cluster. In all cases where a reference implementation exists, Liszt code performs within 16% of the reference code and often outperforms the reference slightly, despite handling all the details of parallelization automatically.

Figure 6 shows the ability of Liszt applications to scale on a large cluster. Compared with our reference implementations of the Euler and Navier-Stokes solvers, the Liszt versions scale similarly.

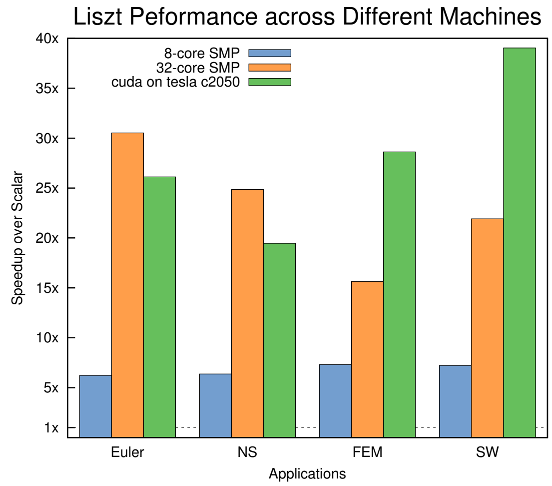


Figure 7: Performance of Liszt applications running on 3 different machines: an 8-core Intel Nehalem E5520, a 32-core Nehalem-EX X7560, and a NVIDIA Tesla C2050. Speedup is measured relative to the scalar reference implementation of each example run on the E5520. Each mesh contained between 200k and 700k elements.

In addition to the SMPs that traditionally compose large clusters, GPUs are an attractive target for computational science due to both their increased efficiency in terms of FLOPS/watt and their price. However, the need to rewrite applications specifically for the GPU has limited adoption. Furthermore SMPs are still desirable since they contain large caches and shared memories and are viewed as easier to program than GPUs. Since Liszt applications are portable, it is easy to use either architecture without modifying application code. We compare the performance of two Intel SMPs with our CUDA implementation on an NVIDIA graphics card in Figure 7. Since SMPs can run both the pthreads and MPI runtime, we report whichever one runs better, for simplicity.

GPU performance shows order-of-magnitude speedups for all applications compared with the scalar reference implementation. In particular we see a 19.5x speedup for the Navier-Stokes solver. Similar fluid-flow solvers in the literature report speedups between 7 and 20x [8, 17]. Asouti et al. report a 46x speedup but only after mixing single- and double-precision arithmetic by hand [3]. Furthermore, Giles et al. report a 3.5x speedup over an 8-core SMP implementation [14]; Liszt shows a similar 3.1x speedup over our own 8-core SMP run. Since each of these experiments was performed on slightly different hardware, direct comparison is difficult; but our results are within the range of performance reported in the literature.

In cases where the working set is large, such as the Euler and Navier-Stokes solvers, the 32-core SMP is able to outperform our GPU runtime. In general, different properties of the applications will make them more suited to a particular architecture. Allowing the programmer to write the programs in an architecture-independent way makes it easier to make these performance comparisons and adapt hardware choices to suit the needs of the application.

These performance results demonstrate that Liszt applications written at high level of abstraction perform comparably to hand-written C++ code, scale on large clusters, and are portable to new parallel architectures such as GPUs.

6. Discussion

We have designed Liszt such that our compiler can automatically determine data dependencies while still being flexible enough to represent different numeric methods for solving PDEs. The elementwise formulation provides a natural level of abstraction to the programmer that is still agnostic of the particular parallel implementation. Furthermore, since most operations at this level are data-parallel operators on all the elements, we can provide parallel *for-comprehension* and *reductions* to reduce the number of dependencies while maintaing the numeric correctness of the program.

Though Liszt infers the data dependencies through the use built-in topological functions, our particular implementation can easily be adapted to work with other methods. Since user-defined relations provide a more natural way to express the degrees of freedom of higher-order elements, allowing the programmer to define additional relations can make the platform more flexible.

For DSLs to be used in large applications, it must be easy to interact with non-DSL code. We plan to address investigate solutions to this issue by creating an interface between Liszt and external sparse matrix solvers.

Reaching exascale computing will require more efficient design. Extreme specialization, like that of D. E. Shaw's Anton [27], allows for orders of magnitude more efficiency but is not backwards compatible with low-level implementations. DSLs such as Liszt separate the specification of the problem from a specific implementation. In addition to providing forward portability, this separation can free hardware designers to make more aggressive changes to hardware. Ideally, the DSL approach for high-performance computing can enable aggressive hardware and software design that will make exascale achievable without needing to rewrite high-level programs for each new architecture.

References

[1] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin. A Comparison of Parallel Graph Coloring Algorithms, 1995.

[2] Corinne Ancourt, Fabien Coelho, and Ronan Keryell. How to Add a New Phase in PIPS: The Case of Dead Code Elimination. In *In Sixth International Workshop on Compilers for Parallel Computers*, 1996.

[3] V. G. Asouti, X. S. Trompoukis, I. C. Kampolis, and K. C. Giannakoglou. Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on graphics processing units. *International Journal for Numerical Methods in Fluids*, pages n/a–n/a, 2010.

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[5] T. Brandvik and G. Pullan. Sblock: A framework for efficient stencil-based PDE solvers on multi-core platforms. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1181 –1188, july 2010.

[6] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 835–847, New York, NY, USA, 2010. ACM.

[7] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Comput. Lang.*, pages 47–57, 1981.

[8] Andrew Corrigan, Fernando Camelli, Rainald Löhner, and John Wallin. Running unstructured grid CFD solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference*, number AIAA 2009-4001, June 2009.

[9] Cray Inc. Cray unveils the cray xk6 supercomputer, May 2011.

[10] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Monserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. In Submission.*, SC '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[11] D. P. Dobkin and M. J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. In *Proceedings of the third annual symposium on Computational geometry*, SCG '87, pages 86–99, New York, NY, USA, 1987. ACM.

[12] John B. Drake, William Putman, Paul N. Swarztrauber, and David L. Williamson. High Order Cartesian Method for the Shallow Water Equations on a Sphere, 1999.

[13] T. Dupont, J. Hoffman, C. Johnson, R. Kirby, M. Larson, A. Logg, and R. Scott. The FEniCS project. Technical report, 2003.

[14] M.B. Giles, G.R. Mudalige, Z. Sharif, G. Markall, and P.H.J Kelly. Performance Analysis of the OP2 Framework on Many-core Architecture. In *ACM SIGMETRICS Performance Evaluation Review (to appear)*, March 2011.

[15] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31:397–423, September 2005.

[16] Anthony Jameson, T.J. Baker, and N.P. Weatherill. Improvements to the aircraft euler method. In *AIAA 25th Aerospace Sciences Meeting*, number 86 - 0103, January 1986.

[17] I.C. Kampolis, X.S. Trompoukis, V.G. Asouti, and K.C. Giannakoglou. CFD-based analysis and two-level aerodynamic optimization on graphics processing units. *Computer Methods in Applied Mechanics and Engineering*, 199(9-12):712 – 722, 2010.

[18] G. Karypis, V. Kumar, and Vipin Kumar. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *Journal of Parallel and Distributed Computing*, 48:71–95, 1998.

[19] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[20] Amy W. Lim and Monica S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Partitions. In *Parallel Computing*, pages 201–214. ACM Press, 1998.

[21] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6:40–53, March 2008.

[22] NVIDIA Corporation. Nvidia tesla GPUs power world's fastest supercomputer, 2010.

[23] OpenCFD Ltd. *OpenFOAM: The Open Source CFG Toolbox User Guide*, August 2010.

[24] R. Pecnik, V. E. Terrapon, F. Ham, and G. Iaccarino. Full system scramjet simulation. *Annual Research Briefs of the Center for Turbulence Research, Stanford University, Stanford, CA*, 2009.

[25] Olivier Pironneau, Frdric Hecht, Antoine Le Hyaric, and Jacques Morice. Freefem, 2005. Universitè Pierre et Marie Curie Laboratoire Jacques-Louis Lions, http://www.freefem.org/.

[26] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.

[27] David E. Shaw, Ron O. Dror, John K. Salmon, J. P. Grossman, Kenneth M. Mackenzie, Joseph A. Bank, Cliff Young, Martin M. Deneroff, Brannon Batson, Kevin J. Bowers, Edmond Chow, Michael P. Eastwood, Douglas J. Ierardi, John L. Klepeis, Jeffrey S. Kuskin, Richard H. Larson, Kresten Lindorff-Larsen, Paul Maragakis, Mark A. Moraes, Stefano Piana, Yibing Shan, and Brian Towles. Millisecond-scale molecular dynamics simulations on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 39:1–39:11, New York, NY, USA, 2009. ACM.

[28] James R. Stewart and H. Carter Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elem. Anal. Des.*, 40:1599–1617, July 2004.

[29] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby. A tensorial approach to computational continuum mechanics using object-oriented techniques. *Comput. Phys.*, 12:620–631, November 1998.