Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers

Zachary DeVito* Niels Joubert* Francisco Palacios† Stephen Oakley‡
Montserrat Medina‡ Mike Barrientos* Erich Elsen‡ Frank Ham‡ Alex Aiken*
Karthik Duraisamy† Eric Darve‡ Juan Alonso† Pat Hanrahan*

* Department of Computer Science, Stanford University {zdevito,niels,mbarrien,aiken,hanrahan}@cs.stanford.edu
† Department of Aeronautics and Astronautics, Stanford University {fpalacios,dkarthik,jjalonso}@stanford.edu
‡ Department of Mechanical Engineering, Stanford University {sjoakley,mmmedina,eelsen,fham,darve}@stanford.edu

ABSTRACT

Heterogeneous computers with processors and accelerators are becoming widespread in scientific computing. However, it is difficult to program hybrid architectures and there is no commonly accepted programming model. Ideally, applications should be written in a way that is portable to many platforms, but providing this portability for general programs is a hard problem.

By restricting the class of programs considered, we can make this portability feasible. We present Liszt, a domain-specific language for constructing mesh-based PDE solvers. We introduce language statements for interacting with an unstructured mesh, and storing data at its elements. Program analysis of these statements enables our compiler to expose the parallelism, locality, and synchronization of Liszt programs. Using this analysis, we generate applications for multiple platforms: a cluster, an SMP, and a GPU. This approach allows Liszt applications to perform within 12% of hand-written C++, scale to large clusters, and experience order-of-magnitude speedups on GPUs.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization, Code Generation*; J.2 [**Physical sciences and engineering**]: Aerospace, Engineering

General Terms

Languages, Design, Performance

Keywords

compiler analysis and program transformations, programming and runtime environments for high performance and high throughput computing

SC11, November 12–18, 2011, Seattle, Oregon USA

1. INTRODUCTION

Modern heterogeneous architectures achieve efficiency through the use of specialized hardware, often optimized for floating point throughput [16, 34]. Several of the world's top supercomputers, such as IBM-LANL Roadrunner [5] and Tianhe-1A [28], already combine multi-core processors with accelerators to achieve petaflop performance. To reach exascale computing, we will need even more power-efficient platforms, which are likely to use heterogenous architectures.

However, programming such systems has proven problematic. A range of competing programming models exist, such as MPI [15], OpenMP [30], and CUDA/OpenCL [22, 26]. These models are difficult to compose, and even though some of these like OpenCL can run on multiple platforms, programs frequently require tuning for each platform. Applications should ideally be written in an environment that makes it possible to target a variety of parallel hardware without requiring significant program modifications. Current general-purpose parallel programming languages are tackling these problems, but it is unclear whether these languages will be successful. Portability is challenging because of the varying parallel programming abstractions exposed by different hardware and the difficulty of automatically analyzing and compiling programs.

In order to perform well on modern hardware, one needs to find parallelism, expose locality, and reason about synchronization. It is difficult to extract this information from general programs, but it is often feasible for restricted classes of programs. For example, a compiler might be able to fully analyze matrix computations in order to target heterogeneous hardware. Our approach is to specialize for a domain by using a high-level language or framework. By embedding domain knowledge in the compiler, we can automatically map code to run on a range of parallel architectures.

We present a domain-specific language, Liszt, for solving partial differential equations (PDEs) on unstructured meshes. These solvers are used in many fields such as computational fluid dynamics or mechanics. Section 2 introduces language statements that capture the data-parallelism of the mesh, the locality of the PDE stencil, and the synchronization of dependencies that occur between phases of an application. In Section 3, we show how this domain knowledge is exploited by program analyses that enable different parallel execution strategies: a partitioning approach based on message passing and a scheduling approach based on graph coloring. In Section 4, we present our implementation of the Liszt compiler, which uses these execution strategies to tar-

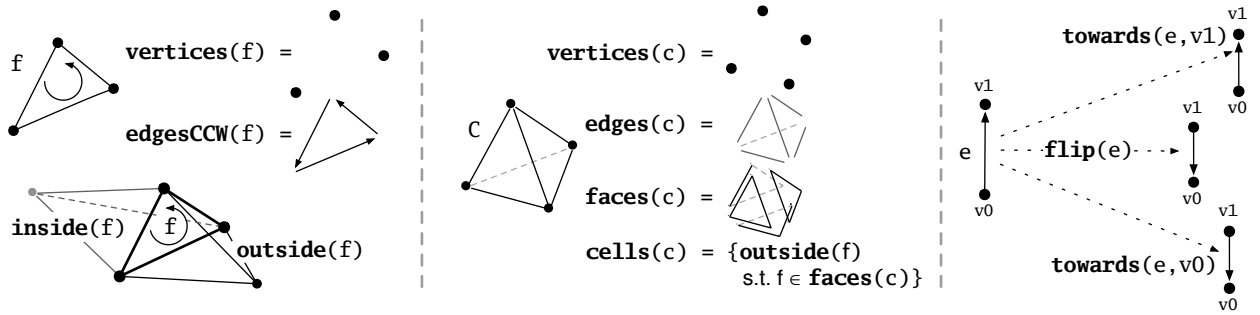


Figure 1: *Liszt's major built-in topological relations.* **outside** *and* **inside** *(the duals of* **head** *and* **tail***) extract the cells on either side of the face. The other duals are also included (e.g.* **edges**(v)*, the dual of* **faces**(c)*).* **flip** *and* **towards** *orient elements.*

```
//Initialize data storage
val Position = FieldWithLabel[Vertex,Float3]("position")
val Temperature = FieldWithConst[Vertex,Float](0.f)
val Flux = FieldWithConst[Vertex,Float](0.f)
val JacobiStep = FieldWithConst[Vertex,Float](0.f)
//Set initial conditions
val Kq = 0.20f
for (v <- vertices(mesh)) {
  if (ID(v) == 1)
    Temperature(v) = 1000.0f
  else
    Temperature(v) = 0.0f
}
//Perform Jacobi iterative solve
var i = 0;
while (i < 1000) {
  for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v2) - Position(v1)
    val dT = Temperature(v2) - Temperature(v1)
    val step = 1.0f/(length(dP))
    Flux(v1) += dT*step
    Flux(v2) -= dT*step
    JacobiStep(v1) += step
    JacobiStep(v2) += step
  }
  for (p <- vertices(mesh)) {
    Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
  }
  for (p <- vertices(mesh)) {
    Flux(p) = 0.f; JacobiStep(p) = 0.f;
  }
  i += 1
}
```

Figure 2: *This Liszt example calculates the temperature equilibrium due to heat conduction on a grid using a Jacobi iteration.*

get runtimes for MPI, pthreads, and CUDA. In Section 5, we present the results of porting a range of applications to Liszt, using our runtimes to target three platforms: a distributed memory cluster, an SMP, and a GPU. We demonstrate that Liszt applications are portable across the wide range of machines and perform comparably to well-tuned hand-written code.

2. LANGUAGE DESIGN

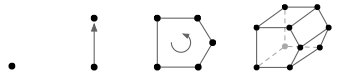
We first describe some strategies for solving PDEs, and then describe the features in Liszt that address these techniques. We assume that the governing PDEs are discretized on a mesh over a region of space. The spatial extent is represented as an unstructured mesh that contains a number of different elements. The PDE itself is discretized to create operators that are used to iteratively compute new values from existing ones. Since the same operators are applied to each element of the mesh, PDE solvers typically expose a large degree of data-parallelism.

Mesh elements are associated with their neighbors through topological relationships. For example, a face separates two cells. Operators derived from the discretization of the PDEs are implemented using these topological relationships. In most cases, the operators are *local*; that is, the computed values depend only on a set of neighboring values called the *stencil*. The extent of the stencil depends on the choice of discretization of the governing PDE, but most existing approaches to the numerical solution of PDEs use a local stencil of the form described here [25]. Data locality of the computation is thus captured by the bounded stencil.

Finally, the iterative application of operators implies that computation occurs in stages. Values of a field are normally read or written, but not both within the same stage; this state of computation we call a *phase*. Data-dependencies occur between coarse-grained phases, but not within the data-parallel implementation of a particular phase. This limits the synchronization necessary for a parallel implementation.

Liszt is designed with this problem formulation in mind. The Liszt programming environment is based on Scala [29] and has support for standard arithmetic types, control flow (**if**- and **while**-statements), and statically-typed non-recursive functions. We extend Scala with types and statements to support the implementation of PDE solvers, while also ensuring that the compiler can reason about parallelization, locality, and synchronization. More specifically, we have added the following capabilities to the language:

Mesh elements Topological entities are first-class abstract data types that describe a discretized 3D manifold of arbitrary polyhedra. We provide a single mesh consisting of vertex, edge, face, and cell elements:



Sets Mesh elements can be grouped into sets. Expressions like **cells**(mesh) refer to all the cells in a mesh, and **vertices**(cell) refer to all the vertices in a cell.

Topological Relationships Given a mesh element, neighboring elements can be accessed only through a suite of built-in topological functions on 3D polyhedra, summarized in Figure 1. For instance, **head**(e) and **tail**(e) return the two vertices on either side of edge e. We construct our topological relations from the operators of Laszlo and Dobkin [11].

Fields Data can be stored on an element using a field. A field is an associative array that maps all mesh elements of a particular type to a value. They are similar in spirit to the fields in Sandia's Sierra framework [35]. Fields can be initialized as constants or from external data.

Parallelism A parallel `for`-comprehension expresses computation for all the mesh elements in a set. We use the term `for`-comprehension rather than `for`-loop since Liszt's `for`-comprehension does not contain any loop-carried dependencies.

Figure 2 shows a simple heat conduction application written in Liszt that calculates the equilibrium temperature distribution on an unstructured grid of rods using a Jacobi iteration. Lines 2–5 construct fields to store the solution. Lines 8–13 use a `for`-comprehension to set up an initial condition with temperature concentrated at one point. Lines 16–35 perform 1000 Jacobi iterations to solve for the steady-state temperature distribution. Each iteration operates on all edges in the mesh, reading the `Position` and `Temperature` fields at the **`head`** and **`tail`** of the edge. Reduction operators are used to calculate new values for the `Flux` and `JacobiStep` at each iteration.

In addition to having useful types for solving PDEs, the language has also been carefully designed so that it can be analyzed and mapped to modern hardware.

2.1 Parallelism

Liszt's `for`-comprehensions expose the data-parallelism in the mesh. The calculations in the body of the statement are independent. We make no assumptions about how the statement is parallelized, so Liszt has the freedom to choose different implementation strategies on different hardware. `for`-comprehensions can appear inline in code, can be nested to arbitrary depth, and match the style of programming typical of our users.

2.2 Locality

The stencil of the PDE captures the pattern of memory accesses of the computation at each element. While there are many different ways to express the stencil, our approach is to approximate it from the use of topological relationships and field accesses using static analysis. To ensure this analysis is precise, Liszt has the following restrictions, which our compiler enforces:

- The topology of the mesh is fixed during execution.
- Mesh elements are accessed only through the use of built-in topological functions. They cannot be arbitrarily constructed using, for instance, integer indices.
- Variable assignments to topological elements and fields are immutable (i.e. any such variable has a single static assignment to a value). This restriction ensures program analysis can accurally bound the set of mesh elements referenced by a variable. In particular, it prevents a program from performing a mesh traversal of unbounded depth which would produce an imprecise stencil.
- Data in fields can only be accessed through the corresponding mesh element.

In Section 3.2, we describe how we automatically determine the stencil from the program using these restrictions.

2.3 Synchronization

Computation occurs in phases: data produced in one phase is used only in subsequent phases. Data-dependencies between phases imply the need for synchronization. Cluster implementations in particular may need to transfer data from the halo of one partition to another between two phases of computation. Liszt codifies these dependencies in the semantics of its field objects.

All fields have a current state, referred to as their *field phase*. At any point in the program, a field is in a single phase: read-only, write-only, or reduce-using-operator `[op]` (only reductions of the form `field(`v`) [op] =` e are allowed). The compiler enforces that a field may not change phase inside the dynamic scope of a `for`-comprehension. Outside of a `for`-comprehension a field can change phase at any time. Limiting fields to a single phase inside a `for`-comprehension restricts intra-phase dependencies while enabling dependencies between coarse-grained phases.

There is one common exception where element-wise calculations have intra-phase dependencies. When calculating the flux in a cell-centered PDE solver, the value in a cell results from the summation of the fluxes calculated at each face. Since flux calculations are expensive operations, it is common to calculate the flux at a face and scatter it to cells that share the face. The scatter operation introduces a dependency among all faces writing to the cell. To address this common case, Liszt also provides commutative and associative *reductions* (eg. `*=`, `+=`) that run atomically. Since the properties of the reduction allow them to be reordered, and since we use the stencil to determine when reductions interfere, we can produce efficient atomic implementations of reductions.

3. PROGRAM ANALYSIS

The domain-specific statements in the Liszt language enable program analyses that determine the stencil and track field phases. We use this analysis to implement two parallel execution strategies for Liszt code: a partitioning-based approach and a coloring-based approach. We first give an overview of these two strategies. We then show how we use the program analyses to implement each strategy.

3.1 Parallel Execution Strategies

The first strategy is suited to distributed memory systems, while the second is suited to architectures with shared memory and a high degree of parallelism.

3.1.1 Partitioning

PDE solvers need to run on large distributed memory machines. Since communication between nodes is significantly more expensive than local operations, synchronization between nodes should be minimized. Applications for these clusters typically use a mesh partitioning approach that minimizes the size of the boundary between partitions [31, 35]. The stencil of elements along the boundary of a partition may require data from non-local elements. These *ghost* elements are duplicated locally. When their values become stale (if, for instance, another node updates the value of its ghost using a reduction), messages are exchanged between nodes to send the updated values. These updates are performed between the computational phases of solvers.

3.1.2 Coloring

Liszt also targets architectures with multiple computational units that access data in a single memory space, such as a GPU. Modern GPUs are multi-core, multi-threaded vector machines that emphasize throughput rather than latency. As an example, the NVIDIA GF100 architecture consists of 15 cores with 48 execution contexts each, where a context consists of a 32-wide vector lane [27]. This design enables 23,040 individual work items to exist simultaneously.

Each piece of work can run independently. However, writes to the same memory location, like those that occur in Liszt's reductions, cause data races.[1] We can schedule work items such that no two items in flight at the same time will perform writes to the same memory location. We employ a graph-coloring approach: an *interference-graph* is built, where two items share an edge if they write to the same value. Coloring this graph such that no two adjacent nodes share the same color ensures that items with the same color can run independently [1, 19]. Work is launched in batches of a single color, separated by barriers. Unlike partitioning, this approach does not require data duplication or communication phases.

3.2 Inferring Stencils and Phases

Both the partitioning and coloring approaches require knowledge of the application's stencil and the locations in the program where fields can change phase. Liszt extracts the stencil and tracks field phases using two domain-specific program analyses.

3.2.1 Stencil

For a given Liszt expression, the stencil specifies which field entries are read and written. To calculate a stencil for a Liszt expression e_l, we use static analysis to create a new expression e_l' that, when run on a particular mesh, generates the reads and writes for the original expression. The expression e_l' has two important properties. First, it *over-approximates* the set of locations that e_l reads and writes. That is, if e_l reads (resp. writes) a location, e_l' is also guaranteed to read (resp. to write) that same location. Second, e_l' is guaranteed to terminate, even if the original expression e_l does not. We first present a definition of the stencil as used in Liszt, and the program transformation that allows Liszt to evaluate it. Section 3.3 then shows how we use the information in the stencil to calculate the ghost elements (for partitioning) and the interference-graph (for coloring).

We represent the stencil as a function, S, that takes a Liszt expression e_l, and an environment E mapping the free variables in e_l to their values. It computes (R, W), where R is a set of field entries that the program will read, and W is the set of field entries that it will write (or reduce) if we execute e_l in the context E:

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

We express the field entry for element v in field f as the pair (f, v).

As an example, consider the result of evaluating the stencil for the flux calculation in the previously mentioned heat conduction example. For a single edge $e_0 = (A, B)$ in the mesh, we find:

$$\mathcal{S}(\texttt{heat\_transfer:18-26}, E\{\texttt{e} \leftarrow e_0\}) = (R, W)$$

where

$$R = \{(\texttt{Position}, A), (\texttt{Position}, B), (\texttt{Temperature}, A), (\texttt{Temperature}, B)\}$$
$$W = \{(\texttt{Flux}, A), (\texttt{Flux}, B), (\texttt{JacobiStep}, A), (\texttt{JacobiStep}, B)\}$$

Since the stencil depends on the execution of a general program, we cannot guarantee that it has some desirable properties, such as always terminating. For this reason, we instead define a conservative approximation, $\overline{\mathcal{S}}$, of the stencil such that for all e_l and E:

$$\mathcal{S}(e_l, E) \subseteq \overline{\mathcal{S}}(e_l, E)$$

where $\subseteq$ is defined point-wise on the pair (R, W).

To implement the approximate stencil, we create an expression e_l' based on e_l such that $\mathcal{S}(e_l', E) = \overline{\mathcal{S}}(e_l, E) \supseteq \mathcal{S}(e_l, E)$, and such that e_l' is guaranteed to terminate. To evaluate $\overline{\mathcal{S}}(e_l, E)$, we can execute the expression e_l' for the environment E, and record the field entries and topological elements it accesses as they occur.

We define an operator $\mathcal{T}$ from Liszt program e_l to e_l' as follows:

$$\mathcal{T}(\ \texttt{if}(e_p)\ e_t\ \texttt{else}\ e_e\ ) \equiv\ \mathcal{T}(e_p);\ \mathcal{T}(e_t);\ \mathcal{T}(e_e);$$
$$\mathcal{T}(\ \texttt{while}(e_p)\ e_b\ ) \equiv\ \mathcal{T}(e_p);\ \mathcal{T}(e_b);$$
$$\mathcal{T}(\ f(e_0,...,e_n)\ ) \equiv\ f'(\mathcal{T}(e_0),...,\mathcal{T}(e_n))$$

where functions f and f' are defined as

$$\texttt{def}\ f(a_0,...,a_n) = e_b\ \text{ and}$$
$$\texttt{def}\ f'(a_0,...,a_n) = \mathcal{T}(e_b)$$

For each remaining kind of expression e, the transform $\mathcal{T}$ is defined by recursively distributing $\mathcal{T}$ to the subexpressions of e.

Note that `if`-statements are transformed such that both branches of the statement execute, which causes strictly more field accesses. Since executing extra code can only add to R' and W' this guarantees that e_l' conservatively approximates e_l, regardless of the path the program takes. Also, `while`-loops are transformed such that the predicate and body of the loop execute exactly once. Since the assignments to mesh variables cannot change from one iteration to the next, it is sufficient to execute the body of the loop once to capture all mesh-access patterns it will perform. Each function call to f is replaced with a call to the function f' that is the result of applying $\mathcal{T}$ to the body of f.

Since Liszt has no looping statements except `while`(e_p) e_b and since `for`-comprehensions iterate over finite sets, the transformed code must terminate when executed. Since each change only increases the size of the resulting sets, $\mathcal{S}(e_l, E) \subseteq \mathcal{S}(e_l', E)$. We optimize our implementation of $\overline{\mathcal{S}}$ by eliminating unnecessary mathematical operations that do not affect the stencil. Furthermore, we use `for`-comprehension fusion and common subexpression elimination on e_l' to improve the efficiency of executing e_l' on a large mesh.

Figure 3 shows the result of applying $\mathcal{T}$ to the heat conduction example. Since $\mathcal{T}$ removes all conditionals, loops,

[1] NVIDIA's Fermi architecture has hardware support for some fast atomic operations, like floating point addition [27], but the lack of double-precision addition limits their usability for physical simulations.

```
val Position = FieldWithLabel[Vertex,Float3]("position")
val Temperature = FieldWithConst[Vertex,Float](_)
val Flux = FieldWithConst[Vertex,Float](_)
val JacobiStep = FieldWithConst[Vertex,Float](_)
for (v <- vertices(mesh)) {
  Temperature(v) = _
  Temperature(v) = _
}
for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  Position(v1) - Position(v2)
  Temperature(v1) - Temperature(v2)
  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}
for (p <- vertices(mesh))
  Temperature(p) += Flux(p)/JacobiStep(p)
for (p <- vertices(mesh)) {
  Flux(p) = _
  JacobiStep(p) = _
}
```

Figure 3: *The result of applying $\mathcal{T}$ to the heat conduction example and removing unnecessary arithmetic operations.*

and arithmetic, the code simply performs mesh operations and field accesses. We have found the approximate stencil to be nearly exact in the programs we examined since solvers rarely have control-flow dependent field accesses.

3.2.2 Field Phase Changes

When the phase of a field changes, a runtime may need to perform an action such as cluster communication. During analysis, Liszt inserts `enterPhase(`p_f`,`f`)` statements into the program that make explicit where field f enters phase p_f. The compiler transforms the program as follows:

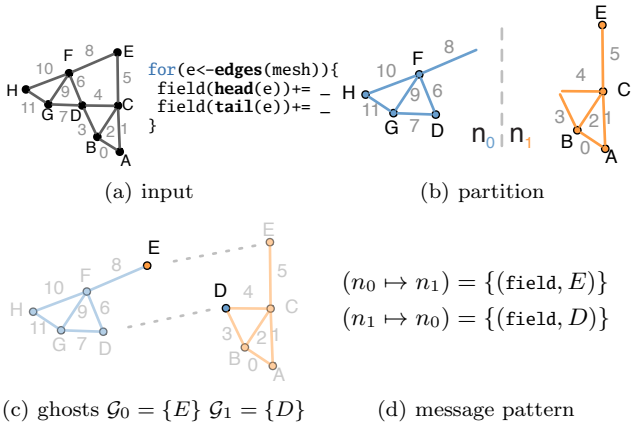
- the statement `a = f(v)` is replaced with `enterPhase(READ,f); a = f(v)`.
- the statement `f(v) [op]= a` is replaced with `enterPhase([op],f); f(v) [op]= a`.
- In each `for`-comprehension `for(`v_i `<-` e_s`)` e_b, we find the set C of all `enterPhase` statements in e_b (and all functions that e_b calls), and replace the `for`-comprehension with C; `for(`v_i `<-` e_s`)` e_b. If C contains two statements, `enterPhase(`p_1`,`f`)` and `enterPhase(`p_2`,`f`)`, such that $p_1 \neq p_2$, then f may change phase within the `for`-comprehension, and Liszt reports a compiler error.

Since, by construction, fields cannot change phase inside a `for`-comprehension, we run a second pass over the program that removes all `enterPhase` statements from any nested code.[2] We perform redundancy elimination on `enterPhase` statements as a dataflow pass over the program. Since this analysis occurs statically, it is conservative. To prevent extra phase changes from invoking cluster communication, we track the phase of a field at runtime and only perform communication when the field undergoes a phase change.

3.3 Implementing Partitioning and Coloring

Given a stencil for a Liszt program, and the program annotated with `enterPhase` statements, we can parallelize it using either the partitioning or coloring strategy from Section 3.1.

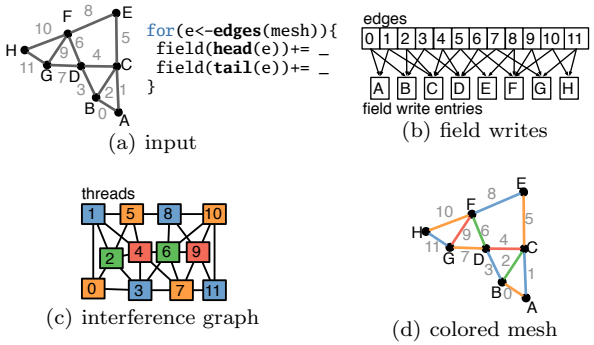
[2]We refer to code that runs outside the dynamic context of a `for`-comprehension as *un-nested* code, and code that runs inside a `for`-comprehension as *nested* code.



(a) input (b) partition

(c) ghosts $\mathcal{G}_0 = \{E\}$ $\mathcal{G}_1 = \{D\}$ (d) message pattern

Figure 4: *The result of applying the partitioning strategy to (a). We generate an initial partition (b) and then detect ghosts (c), which determines the communication pattern (d).*



(a) input (b) field writes

(c) interference graph (d) colored mesh

Figure 5: *The result of applying the coloring strategy to (a). We use the inferred stencil to generate the set of field writes (b) that are performed for each edge. We construct and color an interference-graph (c); each edge of the same color can be evaluated in parallel, as shown in (d).*

3.3.1 Partitioning

To compute a partitioning of Liszt code over n nodes, we first compute an n-way partition of all of the mesh elements. For 3D meshes, we build a graph with edges between two cells that share a face, and use ParMETIS to compute a partitioning of cells [21]. Any lower-dimensional element is heuristically assigned to a partition that contains one of its neighboring cells. While we could use a stencil-derived graph, we have found that this cell-based construction works best when using a heuristic-driven partitioner.

Given an initial partitioning, we then calculate the ghost elements for a partition. Let $\mathcal{D}_n$ be the set of mesh elements assigned to node n. If E is an environment mapping from the set of global mesh variables to their values, then let

$$E_n(v) = E(v) \cap \mathcal{D}_n$$

Intuitively, this operation partitions the global topological values across all nodes, assigning a mesh element to a node only if that node is its owner. Elements that appear in the stencil but are not owned by n are the ghost elements, $\mathcal{G}_n$, for the expression e:

$$\mathcal{G}_n = \{v \,|\, (f, v) \in R'_n \cup W'_n\} - \mathcal{D}_n$$
$$\text{where } (R'_n, W'_n) = \overline{\mathcal{S}}(e_l, E_n)$$

We can compute $\mathcal{G}_n$ by traversing the mesh using the stencil, adding to the set of ghost elements whenever we find an element that is not in $\mathcal{D}_n$.

When using a partitioning-based approach, each node stores the set of locally-accessed topological relationships, and allocates space in each field f for all elements in $\mathcal{D}_n$ and $\mathcal{G}_n$ of the appropriate type.

Figure 4 shows the result of applying this ghost detection to the flux calculation of the heat conduction example. Since the mesh topology does not change, it is sufficient to calculate the ghost elements for the entire program during program initialization.

Since reductions can be reordered within a field phase, Liszt can accumulate intermediates locally rather than immediately sending updates to the owner of the element when performing a reduction on ghosts. When a field f changes from a reduction state to a read state, it is necessary to ensure the most recent values for (f, v) are present on node n if $(f, v) \in R'_n$.

The stencil allows us to precompute the pattern of this communication for each field to eliminate extra buffering. We accomplish this in two steps. First, all deltas for (f, v) are sent to the owner of v, and the owner calculates the updated value of the entry. Second, the new value is sent to all nodes n for which $(f, v) \in R'_n$. Using the mesh-stencil, we can pre-compute the pattern of this communication:

- In the first phase, node n_1 sends n_2 the delta for (f, v) if $(f, v) \in W'_{n_1}$, and $v \in \mathcal{D}_{n_2}$.
- In the second phase, n_1 sends n_2 the new value for (f, v) if $(f, v) \in R'_{n_2}$, and $v \in \mathcal{D}_{n_1}$.

We perform several optimizations to this process to minimize memory usage and communication:

- If for a particular field f: $\forall (f, v) \in W'_n, v \in \mathcal{D}_n$ then we do not allocate memory to store the deltas of f since no operation on n will write to a ghost in f.
- If for a particular field f: $\forall (f, v) \in R'_n, v \in \mathcal{D}_n$ then we do not allocate memory to store the value for ghosts in field f on node n since no operation on n will read from a ghost in f.
- If the message from n_1 to n_2 in either step is found to be empty, then that message is not sent.

The optimizations ensure that fields that are only used locally do not result in any communication, and communication only occurs between nodes that actually share values.

3.3.2 Coloring

When using the coloring approach, a Liszt runtime parallelizes an un-nested `for`-comprehension, `for`$(v_i$ <- $e_s)$ e_b, by finding a non-interfering schedule to run each instance of e_b. We do this by assigning each instance of the `for`-body to a color, such that no two instances in the same color write or reduce to the same field entry. For an instance of a `for`-body, e_b, where $v_i = c$ we can compute the set of field writes for this instance, W'_v, as:

$$(R'_v, W'_v) = \overline{\mathcal{S}}(e_b, E\{v_i \leftarrow c\})$$

We can then create an interference-graph over mesh elements, where the nodes are the set of mesh elements in e_s, and there exists an edge between any two mesh elements (v_1, v_2) if there exists some field entry (f, v) such that

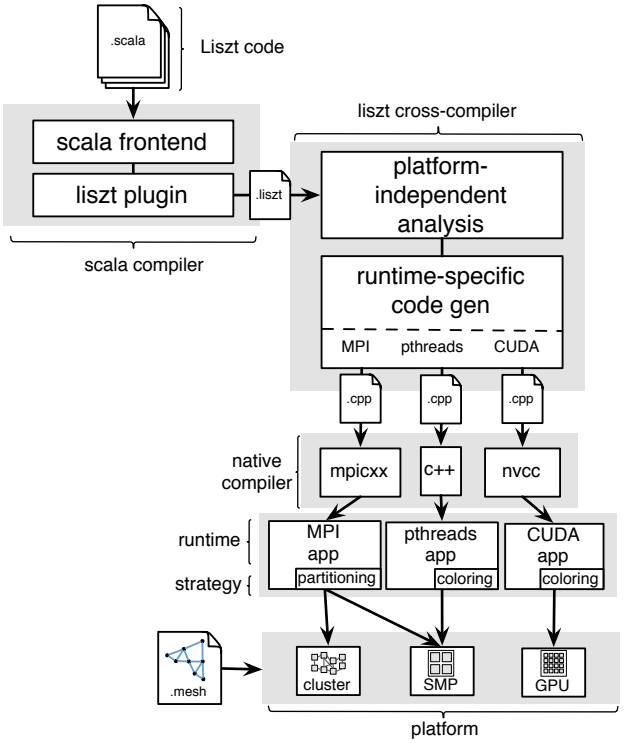


Figure 6: *System diagram for Liszt. Application code is translated to an intermediate representation using a plugin in the Scala compiler. The Liszt compiler then generates native code for multiple runtimes.*

$(f, v) \in W'_{v_1}$ and $(f, v) \in W'_{v_2}$. We compute a heuristically minimal coloring of this interference-graph using Chaitin et al.'s register coloring algorithm [9]. Figure 5 shows the result of applying this coloring algorithm to the flux calculation in the heat conduction example. Since the assignment of values to mesh variables cannot change, this coloring is computed once per static `for`-comprehension at the start of a Liszt program, and reused on subsequent iterations. In our examples, many `for`-comprehensions are trivially independent, as they only write to field entries for element v_i. To reduce initialization time spent in the coloring algorithm, we detect these statements, and assign the entire set to a single color.

4. PLATFORM

Our implementation of Liszt applies these analyses to target three programming models: MPI, CUDA, and pthreads. Figure 6 gives an overview of Liszt's architecture. Liszt runs in three phases. A frontend parses and type-checks Liszt code, and emits an intermediate representation (IR). The Liszt cross-compiler translates this IR into C++ or CUDA code that is designed to compile against one of three runtimes. Finally, a native compiler compiles the C++ and links the runtime code, producing the final executable.

The frontend of Liszt is implemented as a subset of the Scala programming language. We chose Scala as our frontend since its rich implicit type-system and high-level language features like abstract `for`-comprehensions make it a good choice for embedding domain-specific languages [8, 33]. A compiler plugin inserted into the Scala compiler takes an IR of type-checked Scala code, and translates it to a Liszt-specific IR.

The Liszt cross-compiler then performs a series of program transformations before generating code for a specific runtime. We apply the presented static analysis to generate an executable version of $\overline{\mathcal{S}}$, and insert `enterPhase` statements into the code. A second pass then generates code for one of three runtimes based on MPI, CUDA, or pthreads.

During initialization, a Liszt application performs the analyses that require the concrete topology of the mesh: the MPI runtime detects ghost cells and sets up communication patterns for the partitioning strategy, while the CUDA and pthreads runtimes perform coloring.

4.1 MPI

Our distributed memory runtime is built on top of the MPI standard. At initialization, we use the partitioning strategy to calculate a partitioning of the mesh across nodes and discover ghost elements on each node. The cluster runtime is designed to work at large scales, so the mesh is loaded and the stencil is calculated in parallel across all nodes. Since each node loads only part of the mesh topology, we may need to use topological relationships that are not local when discovering ghost elements. To load non-local topology, we apply the stencil to the partition in breadth-first order. At each depth we request the topological relations needed to traverse the next depth from the appropriate nodes in the cluster. This breadth-first process enables the cluster runtime to load meshes that are larger than those that would fit on one node, and it ensures that only the mesh relations used on a node are present there. Each node further improves memory performance by reordering its data into locally contiguous arrays.

Once this setup is complete, the MPI runtime executes using the single-process multiple-data (SPMD) paradigm, with each node performing the same un-nested actions, and performing nested actions on its own partition. Communication between nodes occurs in the `enterPhase` statements when a field changes phase. A straightforward extension to our current implementation would allow this communication to overlap with local computation.

4.2 CUDA

Our CUDA runtime transforms Liszt code into a series of kernel calls. Un-nested code is executed on the host CPU while all nested code is run on the GPU. Each un-nested `for`-comprehension is expressed as a CUDA kernel and kernel invocations. Each free variable in the body is passed as an argument to the kernel. We apply coloring to the `for`-comprehension to determine a valid schedule for each element. The CUDA runtime then launches one kernel per discovered color to execute the `for`-comprehension. Since fields entries can only be accessed from a nested context, we only store them on the GPU, eliminating costly copies. Though our implementation assumes that the entire problem can fit in the GPU's memory, the information provided by the stencil and field phases should allow an implementation to stream data to and from the GPU as necessary.

4.3 pthreads

Our pthreads implementation uses a slight variation of the CUDA runtime: instead of launching kernels, the coloring-based pthreads runtime launches a fixed number of worker threads, which execute un-nested code. `for`-comprehensions are scheduled using coloring, as in the GPU, but rather than compute the interference-graph for single instances of the `for`-body, we compute interference for batches of elements. Grouping elements into batches improves cache coherence since groups of neighboring elements can be executed by the same thread. Each color is then run separated by global barriers. Batches within a color are initially distributed evenly across all worker threads. To prevent work imbalances, a thread will attempt to steal batches from other threads when it finishes its allocation of work.

5. RESULTS

To evaluate the portability and performance of Liszt, we ported four example applications to Liszt and ran these applications on three platforms: a GPU, an SMP, and a cluster. We evaluate the MPI-based runtime on both the cluster and the SMP since it can run on either platform. We further compare the performance of these Liszt applications to their reference implementations. All our runtimes use double precision floating point numbers.

5.1 Example Applications

Our main test applications are solvers for the compressible Euler and Navier-Stokes flow dynamics equations. To show that Liszt can support a broader class of applications, we also implemented a shallow water solver and a finite element code. All four applications have different stencils and arithmetic intensities to test Liszt's ability to produce efficient code for different patterns of data access.

5.1.1 Euler and Navier-Stokes (NS) Solvers

Liszt is developed in cooperation with Stanford University's DOE PSAAP project that aims to quantify the uncertainties in the unstart phenomena in a scramjet engine [31]. The core applications are an Euler and a Navier-Stokes fluid dynamics solver, similar to those widely used in industry. These applications are written in C++ and use MPI to run on large clusters with thousands of nodes.

These solvers calculate the flow through hypersonic propulsion systems (scramjets). Joe is used with mesh sizes in the tens of millions of elements in order to resolve compressibility, turbulence, heat transfer, and combustion physics. Runtimes for typical simulations using approximately 512 cores range from 4–5 hours for a steady-state computation to 2–3 days for unsteady simulations spanning multiple flow-through times. Our cluster-based Liszt experiments were designed to use real-world problem sizes.

The two applications allow testing of different sized working sets, since they use the same stencil but perform different amounts of computation and data accesses. Both applications use cell-centered schemes that store the density, pressure and velocity at each cell centroid of a mesh and solve for these values by performing a face-based flux calculation and a forward Euler time-stepping method. The Euler simulation has a first-order accurate stencil, considering only the values at the two neighboring cells of a face to solve for inviscid flow. The Navier-Stokes solver additionally considers the gradient of values at these cells to calculate both inviscid and viscous flow.

5.1.2 Shallow Water (SW)

This application simulates the height and velocity of the free-surface of the ocean over the spherical earth using an algorithm described by Drake et al [12]. Unstructured tri-

angular mesh elements are used to represent the surface of the sphere. The algorithm uses a staggered formulation with the height of the free-surface being stored at the face-centers and the velocity being stored at the center of the edges. Spatial discretization is performed using a second-order accurate upwind scheme, the stencil of which accesses all edges of the faces on either side of an edge.

5.1.3 Finite Element Method (FEM)

Lastly we show results from a finite element code that computes the solution of the three-dimensional Laplace equation over a mesh of hexahedral elements arranged in a cubic grid using trilinear basis functions. The finite element method leads to a sparse linear system in which the stencil accesses all vertices of an element. The linear system is solved using the conjugate gradient method, with the matrix represented as a field over vertices.

5.2 Scalar Performance

We want to demonstrate that the performance of code written in Liszt is comparable to native C++ code. Liszt code can compile to a canonical serial implementation. Figure 7 presents the results of running the reference scalar code for each application and comparing it to its canonical implementation in Liszt. Each application is iterative and long running, so we measure performance as the average duration of a single iteration. All four examples were executed on a single core of an Intel Nehalem E5520, and we find that the Liszt program performs as well as the equivalent C++.

	Euler	NS	FEM	SW
Mesh	367k cells	668k cells	216k cells	327k faces
Liszt	0.37 s	1.31 s	0.22 s	3.30 s
C++	0.39 s	1.55 s	0.19 s	3.34 s

Figure 7: *Scalar Liszt runs compared to reference C++.*

5.3 Cluster Scaling

For large-scale simulation, Liszt applications must run on large clusters. We use our MPI runtime for this purpose. In this section, we report scaling results for all four applications. In the case of the Euler and Navier-Stokes applications, we also have MPI implementations that run on clusters. We measure speedup compared to the reference code running on 32 cores since the problem is too large to fit on a single node of the cluster.

The cluster used for these experiments consists of 256 nodes, with each node containing two 2.66GHz Intel Nehalem X5650 processors with 16GB of RAM. Scaling tests were performed by varying the number of nodes from 8 through 256 using 4 cores per node for a total of 32 to 2048 cores. The Euler and Navier-Stokes simulations use a 23 million and 21 million cell mesh respectively. In the configurations we present, mesh partitioning runs in less than one minute and only occurs once during program initialization.

Figure 8 shows the scaling results for all four applications. The Liszt implementations all scale linearly to 1024 cores. For the two applications with hand-written MPI C++ implementations, Liszt performs within 12% of the reference implementation. For the Euler simulation, Liszt scales slightly better than the reference C++ code because the reference application's framework hard-coded communication of two fields that were never used. Liszt discovered only the fields necessary for the algorithm and communicated less data. The shallow water application has a knee in the scaling graph at 64 cores, where the specific partitioning caused an increase in the number of ghost cells. The FEM application experiences super-linear scaling from 256 to 512 cores, as the working set of the algorithm becomes small enough to fit entirely into L3 cache.

We notice that both the Liszt and C++ Euler and Navier-Stokes applications stop scaling after 1024 cores. Scaling performance on a cluster is limited by the amount of data per node, since the communication across boundaries limits performance if too little computation occurs in the interior of the partition. The typical inflection point for the PSAAP solvers occur at 20,000 cells per node. The 20 million cell meshes used for these runs causes this inflection point to occur at 1024 cores, which explains the scaling limitations of this experiment. Given a larger mesh we expect these applications to continue scaling.

5.4 GPU Performance

The price-performance ratio of modern GPUs make them attractive to computational scientists. Unfortunately, applications need to be rewritten to target GPUs. Liszt applications can run on this platform using our CUDA runtime.

All four applications were executed using CUDA 3.2 on an NVIDIA Tesla C2050, using the same meshes as for scalar testing. Figure 9 shows that all four applications experienced order-of-magnitude performance gains over scalar implementations, with the Euler solver showing 26.1x and the Navier-Stokes solver 19.5x compared to scalar code on the Nehalem machine. Coloring during program initialization took no more than 16 seconds and required no more than 6 colors using an untuned coloring implementation. For faster startup, a more efficient coloring implementation like that of Giles et al. [14] could be used.

To place this performance into context, we compare the performance of the Navier-Stokes implementation in Liszt to work done on GPUs by other groups. Corrigan et al. report 7.4x over scalar [10], while Kampolis et al. [20] show a 19.6x speedup for their Navier-Stokes solver. Asouti et al. implement a vertex-based method that mixes single and double precision arithmetic to achieve 46x over a scalar implementation [3]. The OP2 project compares their Navier-Stokes solver to an 8-core SMP, achieving a 3.5x speedup [14]. For comparison, we show a 3.1x speedup over our 8-core results in Section 5.5. All these results use various modern GPUs. Since these results were run on different experimental setups, and Navier-Stokes solvers vary, direct comparisons are difficult. Nevertheless, Liszt performance results are within the range of results reported in the literature. The flexibility of Liszt should allow us to adopt GPU execution techniques as they are developed.

To get a sense of the efficiency of our GPU runtime, we investigated how it uses the resources of the GPU. We studied in detail the inviscid flux calculation, which takes the majority of the Euler application's runtime. This loop uses the full 63 registers per thread on the GPU to support the relatively large working set of 17 doubles (the viscous kernel found in the Navier-Stokes simulation uses even more, at 32 doubles). The high register usage limits the warp occupancy on each core. Hardware profiling reveals that the current occupancy reduces our main memory bandwidth to 95GB/s, 66% of peak. Attempts to decrease register usage to increase occupancy reduces performance due to spilling.

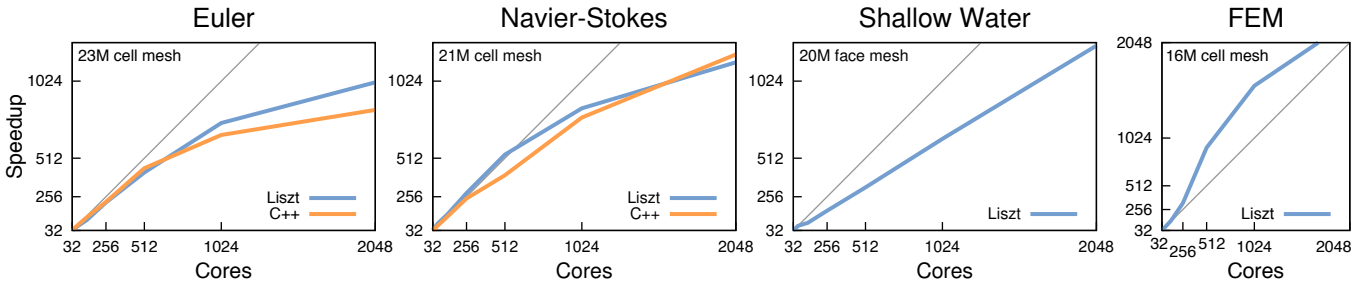


Figure 8: *Scaling four Liszt applications on a 2048-core cluster. Speedup is measured relative to a reference implementation run on 32 cores. The Euler and Navier-Stokes applications are compared to hand-tuned reference implementations. The Shallow-Water and FEM implementations are compared to Liszt-based references since no hand-tuned MPI code exists.*

Furthermore, coloring scatters memory access. Since consecutive memory locations are not necessarily spatially or temporally accessed, overfetch causes the effective bandwidth of this kernel to be 38GB/s. Liszt does as well as other work in the field, even though the memory-bound nature of this kernel limits our overall performance. We conclude that more research is needed to produce an optimal GPU runtime.

We will briefly compare the GPU performance results to our multicore runtime in the next section.

	Euler	Navier-Stokes	FEM	SW
Runtime	0.0141s	0.0673s	0.0076s	0.0842s
Speedup	26.1x	19.5x	28.6x	39.0x

Figure 9: *GPU performance against scalar, using the same meshes as our scalar comparison runs.*

5.5 SMP Performance

	8-core		32-core	
	Euler	NS	Euler	NS
C++ MPI	0.062 s	0.239 s	0.014 s	0.061 s
Liszt MPI	0.060 s	0.206 s	0.014 s	0.058 s
Liszt pthreads	0.059 s	0.218 s	0.053 s	0.168 s

Figure 10: *Single-machine performance, comparing the time-per-iteration of Liszt and reference C++.*

Intel and AMD have aggressively increased the parallelism of their SMP offerings. These multicore machines with shared memory and large caches are attractive, since they support multiple programming models and are viewed as more flexible than GPUs. There is no consensus yet on the ideal way to execute scientific code on these machines. We show that Liszt can target SMPs using either our pthreads or MPI runtime by running all our example applications using both these runtimes, measuring speedup over our scalar runtime.

We use two different SMPs for these runs. We use an 8-core Intel Nehalem E5520-based machine as a reasonable approximation of machines currently in use by scientists writing solvers. We also use a 4-processor 32-core Intel Nehalem-EX X7560 machine, which is representative of the latest server offerings from Intel. SMP results use the same meshes as the scalar result.

First we consider the performance of the Liszt MPI runtime on this platform in comparison to the hand-written MPI implementations we have for the Euler and Navier-Stokes application. We show in Figure 10 that Liszt performs as well as hand-written code on both the 8-core and 32-core machine.

Next we compare the pthreads and MPI runtimes across all four applications. We find that both runtimes show near

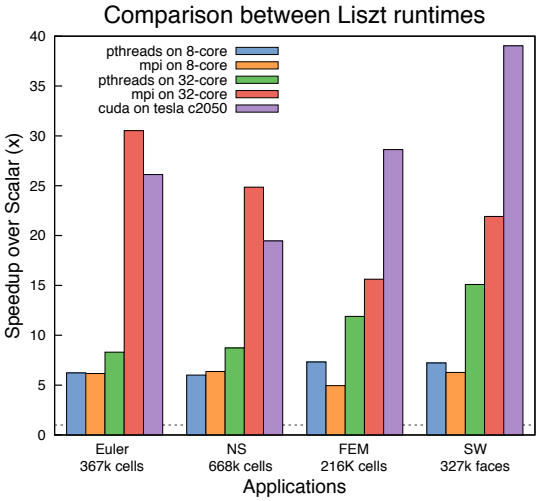


Figure 11: *Performance comparison of 3 Liszt runtimes. Speedup is measured relative to the reference code run on a single core of the machine's CPU.*

linear speedups on an 8-core Nehalem. Figure 11 shows the pthreads runtime outperforming the MPI runtime for our Shallow Water and FEM examples on this machine. Profiling the FEM example, we found that the partitioning approach spent 15% of its runtime waiting for messages from other partitions due to load imbalance, with similar figures for Shallow Water. The pthreads runtime performs better, since its work-stealing approach can balance the load amongst the cores.

On the 32-core Nehalem-EX, the MPI runtime scales linearly to 32 cores while the pthreads runtime peaks at 16 cores. Cache profiling indicates that the careful partitioning used by the MPI runtime achieves a 15% L2 miss rate, while the coloring approach of the pthreads runtime experiences almost three times more L2 misses with a 40% L2 miss rate, explaining the performance differences. We conclude that an ideal way of running on an SMP would combine the careful partitioning of the MPI runtime with the load-balancing aspects of the pthreads runtime. This remains as future work.

Figure 11 summarizes the results presented for all four applications running on the SMP using both runtimes and running on a GPU. The GPU is a clear winner in comparison to the smaller 8-core Nehalem, while the 32-core Nehalem-EX machine managed to outperform the GPU for the Euler and Navier-Stokes simulation. These two codes depend on a large working set per thread, and at 32 cores the larger caches of the Nehalem architecture equalized the differences in memory bandwidth between these platforms.

6. RELATED WORK

Many libraries and frameworks for writing mesh-based PDE solvers are similar to Liszt. However, each addresses the parallelism, locality, and communication of applications in different ways.

To address parallelism, frameworks such as PETSc [4] or Sandia's Sierra framework [35] use a SPMD model where the unstructured mesh is partitioned across discrete memory spaces. This approach enables the programmer to write an application where the majority of the code resembles a single-threaded implementation, and parallelism is only considered when data crosses partitions.

A second approach, taken by OP2 [14] and SBLOCK [6], expresses element-wise computations as kernels applied to a set of data. This design gives the library freedom to choose how to parallelize the operation, but requires reformulating code into explicit kernels and kernel invocations.

Like kernel-based approaches, Liszt's `for`-comprehension provide the freedom to choose a parallel implementation, but like the SPMD-style approaches, Liszt's `for`-comprehensions follow the more familiar style of serial code.

To expose locality, other frameworks also reason about the PDE's stencil. SBLOCK, which implements structured grids on clusters or GPUs, allows the user to declare the stencil as constant-valued offsets, and uses it to schedule kernels [6]. The Overture library for structured grids and ParFUM allow declaring the depth of ghost cells necessary, and automate their setup [7, 23]. In OP2, the stencil is explicitly built as sets of inputs and outputs [14].

Other approaches infer the stencil from the specification of the problem, which reduces the effort of the developer. In structured solvers, the indices of the accessed data are typically affine transformations of the index of the element. In these cases, automatic techniques using affine and polyhedral analysis can detect the stencil and parallelize loops [24]. The PIPS compiler follows this approach [2]. Liszt takes a similar approach in spirit to affine partitioning, but uses the logic of mesh-topology rather than affine transformations to automate the analysis. This approach allows Liszt to work on unstructured meshes, where mesh neighborhoods cannot be expressed with affine indices.

To address synchronization, frameworks that use the SPMD-style approach frequently require explicitly invoked communication to update values in the ghost cells [4, 35]. Forgetting these update statements leads to subtle numerical errors. Kernel-style approaches such as OP2 [14] or SBLOCK [6] can perform this communication implicitly at the beginning or end of kernel invocations.

Another approach is to express the PDEs at the mathematical level. OpenFOAM [36], Sandia's Sundance [17], FreeFEM [32] and FEniCS [13] use a top level of abstraction that expresses the problem in terms of actual differential equations, leaving the details of the parallel implementation to a lower-level library. While this declarative abstraction can be convenient for well-established methods, developing new methods often requires more direct control over the solver algorithm.

7. CONCLUSION

We have presented the Liszt language for writing PDE solvers. Liszt programs are written using high-level operations on mesh elements and fields. The language has been carefully designed so that our compiler can infer the PDE's stencil and the data-dependencies between fields. With this information, we can automatically target different parallel programming models, in particular, MPI, pthreads, and CUDA; and therefore run on different architectures (clusters, SMPs and GPUs). Programs written in Liszt are portable, yet run as efficiently as hand-written code.

We believe that writing code at a higher level increases the productivity of programmers. Programmer productivity is difficult to measure without user studies. However, we have compared the Liszt implementation of the Navier-Stokes fluid flow solver to our reference implementation. The original application consists of 40k lines of code, of which only 2k lines (5%) were needed to implement the algorithm in Liszt. Over 95% of the code is reusable libraries and platform-specific code; refactoring the code to use a higher-level language or framework would eliminate duplicated infrastructure. Also, since Liszt infers stencils and dependencies automatically, we can guarantee the correctness of the parallel implementation, making programs easier to debug.

Our results suggest several future research areas. We are developing the ability to compose our runtimes using an approach similar to that of Houston et al. [18]. Composable runtimes would allow Liszt to run on a cluster of GPU-accelerated machines.

Currently Liszt supports only a small subset of applications. We plan to support more applications in two ways. First, we plan to add additional features to the language. We have designed an extension to the language to support implicit methods and linear solvers. We have also designed more general ways to represent fields; these enhancements to fields are needed to support certain finite-element formulations, such as discontinuous Galerkin. Secondly, we plan to make it possible to embed Liszt code in general-purpose languages. A application can implement a portion of the program in Liszt, and use a general-purpose language for portions of the program that Liszt does not support.

The emergence of heterogenous parallel architectures with different hardware abstractions will make it even more challenging in the future to write portable programs. We have shown that a domain-specific programming environment may be a viable approach for achieving portability and performance. However, multiple DSLs many be required to cover the range of possible applications that need to be run on future machines. For this approach to be successful, in addition to building specific high-level frameworks, we envision new tools that will allow domain experts to build DSLs for their problems much like they build libraries today.

8. ACKNOWLEDGMENTS

The authors wish to thank Jeffrey Fike, Patrice Castonguay, and Steve Jones for providing meshes and technical support. This work was supported by grants from the DoE's Predictive Science Academic Alliance Program; Stanford Pervasive Parallelism Laboratory affiliates program supported by NVIDIA, Oracle, AMD, and NEC; and hardware donations from NVIDIA and Intel. Niels Joubert is supported by a Reed-Hodgson Stanford Graduate Fellowship. The NSF MRI-R2 grant—Acquisition of a Hybrid CPU/GPU and Visualization Cluster for Multidisciplinary Studies in Transport Physics with Uncertainty Quantification, funded under the American Recovery and Reinvestment Act of 2009 (Public Law 111-5)—provided resources for computation.

9. REFERENCES

[1] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin. A comparison of parallel graph coloring algorithms. Technical report, SCCS-666, Northeast Parallel Architectures Center at Syracuse University, 1995.

[2] C. Ancourt, F. Coelho, and R. 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*, 2010.

[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. 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] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of Roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, Piscataway, NJ, USA, 2008. IEEE Press.

[6] 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.

[7] D. L. Brown, G. S. Chesshire, W. D. Henshaw, and D. J. Quinlan. OVERTURE: An object-oriented software system for solving partial differential equations in serial and parallel environments. In *PPSC'97*, 1997.

[8] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. 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.

[9] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Comput. Lang.*, pages 47–57, 1981.

[10] A. Corrigan, F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid CFD solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference*, number AIAA 2009-4001, June 2009.

[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] J. B. Drake, W. Putman, P. N. Swarztrauber, and D. L. Williamson. High order cartesian method for the shallow water equations on a sphere. Technical report, TM-2001, Oakridge Nation Laboratory, 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. Giles, G. Mudalige, Z. Sharif, G. Markall, and P. Kelly. Performance analysis of the OP2 framework on many-core architecture. In *ACM SIGMETRICS Performance Evaluation Review (to appear)*, March 2011.

[15] W. Gropp, S. Huss-Ledermanand, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, 1998.

[16] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. *SIGARCH Comput. Archit. News*, 38:37–47, June 2010.

[17] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31:397–423, September 2005.

[18] M. Houston, J.-Y. Park, M. Ren, T. Knight, K. Fatahalian, A. Aiken, W. Dally, and P. Hanrahan. A portable runtime interface for multi-level memory hierarchies. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 143–152, New York, NY, USA, 2008. ACM.

[19] A. Jameson, T. Baker, and N. Weatherill. Improvements to the aircraft Euler method. In *AIAA 25th Aerospace Sciences Meeting*, number 86 - 0103, January 1986.

[20] I. Kampolis, X. Trompoukis, V. Asouti, and K. 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.

[21] G. Karypis, V. Kumar, and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–95, 1998.

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

[23] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. Kale. ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22:215–235, 2006.

[24] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. In *Parallel Computing*, pages 201–214. ACM Press, 1998.

[25] R. Löhner. *Applied Computational Fluid Dynamics: An Introduction Based on Finite Element Methods*. Wiley, Fairfax, Virginia, 2nd edition, 2008.

[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6:40–53, March 2008.

[27] NVIDIA Corporation. NVIDIA's next generation compute architecture: Fermi, November 2009.

[28] NVIDIA Corporation. NVIDIA Tesla GPUs power world's fastest supercomputer, 2010.

[29] M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. Mcdirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the Scala programming language (second edition. Technical report, LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne, 2006.

[30] OpenMP Architecture Review Board. *OpenMP: Application Program Interface 3.1*, July 2011.

[31] 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.

[32] O. Pironneau, F. Hecht, A. L. Hyaric, and J. Morice. FreeFEM, 2005. Universitè Pierre et Marie Curie Laboratoire Jacques-Louis Lions, http://www.freefem.org/.

[33] T. Rompf and M. 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.

[34] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. 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.

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

[36] 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.