Liszt

Programming Mesh-based PDE Solvers for Heterogeneous Parallel Platforms

Z DeVito, N Joubert, F Palacios, S Oakley, M Medina, M Barrientos, E Elsen, F Ham, A Aiken, K Duraisamy, E Darve, J Alonso, P Hanrahan
Today:

Compiling code for parallel machines

Language design that enables code compilation for parallel machines
  Abstracting away machine details
  Inferring data accesses

Implementing transformations and analysis
  Abstract analysis at compile time
  Build concrete data structures at runtime

Running code on different parallel machines
  Partitioning, Coloring
  Using analyses to target these approaches
The Future is Now

LANL IBM Roadrunner
  (Opteron + Cell)
Tianhe-1A
  (Xeon + Tesla M2050)
ORNL Titan
Why?

1. Specialization leads to efficiency (performance/Watt)
   - Sequential cores optimized for hiding latency
   - Throughput cores optimized for delivering FLOPs
   - Special hardware for compression-decompression, etc.
   - Laptops need to run graphics applications

2. Hybrid architectures more efficient for complex workloads
   - Applications have both task- and data-parallelism
   - Optimal platform has mixture of optimized units
   - Modern version of Amdahl’s Law
Different Technologies at Different Scales

Cluster of SMPs – racks

- Distributed memory over system area network

Small number of CMPs – boards

- Shared memory using chip interconnect

CMP - chips

- Multi-core for sequential performance
- Many-core for throughput performance
- On-chip network
Naturally Different Programming Models

Cluster of SMPs – racks
- *Message passing* - MPI

Small number of Hybrid CMPs – boards
- *Threads and locks* – pthreads, OpenMP

Hybrid CMP – chips
- *GPU cores* – CUDA/OpenCL?
- *Cores with vector instructions* – ArBB??
- *Task queues for scheduling work between CPU/GPU* - Gramps???
How Do We Execute on These Machines?
Execution Strategies

Partitioning
- Assign partition to each computational unit
- Use **ghost** elements to coordinate cross-boundary communication.
- Ideal for single computational unit per memory space

Coloring
- Calculate interference between work items on domain
- Schedule work-items into non-interfering batches
- Ideal for many computational units per memory space
How Do We Program These Machines?
Let the compiler help us.
Compiling to parallel computers

Find Parallelism

Expose Data Locality

Reason about Synchronization
Analyzing data dependencies

“What data does this value depend on”

Find Parallelism
- Independent data can be computed in parallel

Expose Data Locality
- Partition based on dependency

Reason about Synchronization
- Don’t compute until the dependent values are known
What does \texttt{avg[i]} depend on?

```c
int num_id = count_idx();
int* id = read_idx();
int* value = read_values();
int* data = read_data();
int* avg = malloc(num_id * sizeof(int))

for (int i = 0; i < num_id; i++) {
    for (int j = id[i]; j < id[i+1]; j++) {
        avg[i] += data[j];
    }
}
```

Can't be done by compilers \textit{in general}!
\texttt{A[i] = B[f(i)]} – must compute \( f(i) \) to find dependency
What does avg[cell] depend on?

```
int num_id = count_idx();
int* id = read_idx();
int* value = read_values();
int* data = read_data();
int* avg = malloc(num_id * sizeof(int))

for (cell <- cells(mesh)) {
    for (vertex <- vertices(cell)) {
        avg[cell] += data[vertex];
    }
}
```

Avg[cell] depends on data[] for all vertices connected to this cell. We assume this will be a small subset of data[]. Encode this!
Trade off generality
EDSLs solve the dependency problem

Liszt provides domain specific language features to solve the dependency problem:

- Parallelism
- Data Locality
- Synchronization

For solving PDEs on meshes:

All data accesses can be framed in terms of the mesh
Example: Heat Conduction on Grid

```scala
val Position = FieldWithLabel[Vertex, Float3]("position")
val Temperature = FieldWithConst[Vertex, Float](0.0f)
val Flux = FieldWithConst[Vertex, Float](0.0f)
val JacobiStep = FieldWithConst[Vertex, Float](0.0f)

var i = 0;
while (i < 1000) {
  for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)
    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
}
```
Features of high performance PDE solvers

Find Parallelism
- Data-parallelism on mesh elements

Expose Data Locality
- PDE Operators have local support
- Stencil captures exact region of support

Reason about Synchronization
- Iterative solvers
- Read old values to calculate new values
Liszt Language Features

Mesh Elements
- Vertex, Edge, Face, Cell

Sets
- cells(mesh), edges(mesh), faces(mesh), ...

Topological Relationships
- head(edge), vertices(cell), ...

Fields
- val vert_position = position(v)

Parallelism
- forall statements: for( f <- faces(cell) ) { … }
How do we infer data accesses from Liszt?

“Stencil” of a piece of code:

Captures just the memory accesses it performs

Infer stencil for each for-comprehension in Liszt
Language Features for Parallelism

\begin{verbatim}
for (e <- edges(mesh)) {
    ...
}
\end{verbatim}

Data-parallel \texttt{for}-comprehension

\begin{itemize}
\item Calculations are independent
\item No assumptions about how it is parallelized
\item Freedom of underlying runtime implementation
\end{itemize}
Language Features for Locality

Automatically infer stencil (pattern of memory accesses at element)

Restrictions:

- Mesh elements only accessed through built-in topological functions; \texttt{cells(mesh)}, …
- Variable assignments to topological elements and fields are immutable; \texttt{val v1 = head(e)}
- Data in fields can only be accessed using mesh elements \texttt{JacobiStep(v1)}
- No recursive functions
Language Features for Synchronization

Phased usage of Fields

- Fields have field phase state
  - read-only, write-only, reduce-using-operator `field(el) [op]= value`
- Fields cannot change phase within for-comprehension

Associative Operators

- Allow single expensive calculation to write data to multiple elements
- Provide atomic scatter operations to fields
  - e.g. `field(el) += value`
- Introduce write dependencies between instances of for-comprehension
Architecture
Domain Specific Transform: Inferring Stencils through Stencil Detection

Analyze code to detect memory access stencil of each top-level for-all comprehension

- Extract nested mesh element reads
- Extract field operations
- Difficult with a traditional library

```scala
for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)
  val step = 1.0f/(length(dP))
  Flux(v1) += dT*step
  Flux(v2) -= dT*step
  JacobiStep(v1) += step
  JacobiStep(v2) += step
}
```
Domain Specific Transform: Inferring Stencils through Stencil Detection

\[ S(e_l, E) = (R, W) \]

\( e_l \) Expression

\( E \) Environment mapping free variables to values

```java
for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)
    val step = 1.0f/(length(dP))
    Flux(v1) += dT*step
    Flux(v2) -= dT*step
    JacobiStep(v1) += step
    JacobiStep(v2) += step
}
```
Domain Specific Transform:
Inferring Stencils through Stencil Detection

Problem: Don’t know the mesh at compile time!

Break up inference into:

**Static part**: Abstractly reason about operators.

- generate C++ code that queries mesh to build the specific stencil for each for-comprehension

**Dynamic part**: Concretely build data structures by analyzing mesh

- run C++ code on mesh

Anything that directly depends on mesh becomes generated code, executed at runtime.

This means: It is possible to write low-level C++ code that targets our back-end. But ugly and hard!
Implementing stencil detection
The way that won’t work

\[ S(e_l, E) = (R, W) \]

1. Sandbox code by having temporary fields and variables
2. Have every field read and write log the current stack of mesh elements
3. Run the code!

```scala
for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)
    val step = 1.0f/(length(dP))
    Flux(v1) += dT*step
    Flux(v2) -= dT*step
    JacobiStep(v1) += step
    JacobiStep(v2) += step
}
```

Cannot guarantee termination
Can run very long
Must do all the math beforehand. In single core.
Implementing stencil detection

Abstract Interpretation

“Partial execution of a computer program to gain insight into it’s semantics”. Allows us to calculate an approximate stencil

\( T \)

Apply transformation \( T \) to Liszt code

generate code with desirable properties (terminates, fast)
Implementing stencil detection
Abstract Interpretation

“Partial execution of a computer program to gain insight into it’s semantics”. Allows us to calculate an approximate stencil

Apply transformation $\mathcal{T}$ to Liszt code

generate code with desirable properties (terminates, fast)

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

```scala
for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)
  val step = 1.0f/(length(dP))
  Flux(v1) += dT*step
  Flux(v2) -= dT*step
  JacobiStep(v1) += step
  JacobiStep(v2) += step
}
```

```scala
for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)
  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}
```
Implementing stencil detection

Abstract Interpretation

Defining $\mathcal{T}$

\[
\mathcal{T}( \text{if}(e_p) \ e_t \ \text{else} \ e_e) = \mathcal{T}(e_p); \mathcal{T}(e_t); \mathcal{T}(e_e);
\]

Conservatively evaluate if-statements

\[
\mathcal{T}( \text{while}(e_p) \ e_b) = \mathcal{T}(e_p); \mathcal{T}(e_b);
\]

Single Static Assignment of Mesh Variables

\[
\mathcal{T}( f(a_0, \ldots, a_n)) = f'(\mathcal{T}(a_0), \ldots, \mathcal{T}(a_n))
\]

Recursively apply to functions

Everything else, recursively apply to subexpressions of expression
In pictures
Domain Specific Transform: Stencil Detection

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}
...

![Diagram of a mesh with labeled edges and vertices]
for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)

    Flux(v1) += _
    Flux(v2) -= _
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}
Domain Specific Transform: Stencil Detection

for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)

    Flux(v1) += _
    Flux(v2) -= _
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}

...
Domain Specific Transform: Stencil Detection

```scala
for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)

    Flux(v1) += _
    Flux(v2) -= _
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}
...
Domain Specific Transform: Stencil Detection

```scala
for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}
...
```
Domain Specific Transform: Stencil Detection

for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)

    Flux(v1) += _
    Flux(v2) -= _
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}
...

![Diagram of mesh with nodes and edges labeled with numbers, showing the flow of data from one node to another.](image)
Domain Specific Transform:
Stencil Detection

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}
...

Domain Specific Transform:
Stencil Detection

for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)

  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}

...
Domain Specific Transform: Stencil Detection

```scala
for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)
  Flux(v1) += _
  Flux(v2) -= _
  JacobiStep(v1) += _
  JacobiStep(v2) += _
}
...
Domain Specific Transform: Stencil Detection

```
for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)

    Flux(v1) += _
    Flux(v2) -= _
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}
...
```
Domain Specific Transform: Stencil Detection

for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)

    Flux(v1) += _
    Flux(v2) -= _
    JacobiStep(v1) += _
    JacobiStep(v2) += _
}
...

Domain Specific Transform: Stencil Detection

```cpp
for (p <- vertices(mesh)) {
    Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
}
for (p <- vertices(mesh)) {
    Flux(p) = 0.f; JacobiStep(p) = 0.f;
}
```
Domain Specific Transform: Stencil Detection

```c
foreach (p <- vertices(mesh)) {
    Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
}

foreach (p <- vertices(mesh)) {
    Flux(p) = 0.f; JacobiStep(p) = 0.f;
}
```
Domain Specific Transform: Stencil Detection

```cpp
for (p <- vertices(mesh)) {
    Temperature(p) += Flux(p)/JacobiStep(p)
}
for (p <- vertices(mesh)) {
    Flux(p) = 0.f; JacobiStep(p) = 0.f;
}
```
Domain Specific Transform: Stencil Detection

for (p <- vertices(mesh)) {
    Temperature(p) += 0.01f*Flux(p)/JacobiStep(p)
}
for (p <- vertices(mesh)) {
    Flux(p) = _; JacobiStep(p) = _;
}
MPI: Partitioning with Ghosts

1. Partition Mesh (ParMETIS, G. Karypis)
MPI: Partitioning with Ghosts

2. Find used mesh elements and field entries using stencil data and duplicate locally into “ghost” elements.

Implementation directly depends on algorithm’s access patterns.
MPI: Partitioning with Ghosts

3. Annotate for-comprehensions with field preparation statements

```java
Flux.ensureState<LISZT_SUM>();
JacobiStep.ensureState<LISZT_SUM>();
Position.ensureState<LISZT_READ>();
Temperature.ensureState<LISZT_READ>();
for (e <- edges(mesh)) {
    val dP = Position(v1) - Position(v2)
    ...
    Flux(v1) += dT*step
    JacobiStep(v1) += step
}
Temperature.ensureState<LISZT_SUM>();
Flux.ensureState<LISZT_READ>();
JacobiStep.ensureState<LISZT_READ>();
for (p <- vertices(mesh)) {
    Temperature(p) += 0.01f * Flux(p)/JacobiStep(p)
}
...
```
MPI: Partitioning with Ghosts

4. MPI communication is batched during for-comprehensions and only transferred when necessary
Applying Program Analysis: Results

\[
\text{for}(f \leftarrow \text{faces(mesh)}) \{ \\
\quad \text{rhoOutside}(f) := \\
\quad \quad \text{calcflux}(f, \text{rho(outside}(f)))) \\
\quad \quad + \text{calcflux}(f, \text{rho(inside}(f)))
\}
\]
Applying Program Analysis: Results

\[
\text{for } f < \text{faces(mesh)} \}
\]
\[
\rho_{\text{Outside}}(f) := \text{calc}\_\text{flux}(f, \rho(\text{outside}(f))) + \text{calc}\_\text{flux}(f, \rho(\text{inside}(f)))
\]

Ghost Cells
Applying Program Analysis: Results

\[
\text{for } \{ f \} \text{ faces(mesh)}
\]

\[
\rho_{\text{Outside}}(f) := \text{calc_flux}(f, \rho(\text{outside}(f))) + \text{calc_flux}(f, \rho(\text{inside}(f)))
\]

\`
Ghost Cells
``
GPU: Schedule threads with coloring

- Shared Memory
- Field updates need to be atomic
- Concerns about MPI approach – volume vs surface area

Build a graph of interfering writes:
GPU: Schedule threads with coloring

Compile time: Generate code to create field write structure

Runtime: Build this structure using mesh and stencil

```
FORALL_SET(e,edges(mesh))
    Vertex v_1 = head(e);
    igraph->addEdge(thread(e).ID(), v_1.ID(),18886);
    Vertex v_2 = tail(e);
    igraph->addEdge(thread(e).ID(), v_2.ID(),18886);
ENDSET
```
GPU: Schedule threads with coloring

Threads 1 edge assigned to 1 thread

Force Field: Memory

A B C D E F G H
GPU: Schedule threads with coloring
GPU: Schedule threads with coloring

Convert each for-comprehension into a GPU kernel, launched multiple times for sets of non-interfering iterations.

```cpp
__global__ for_01(ColorBatch batch, Force, Position, Velocity, Maxforce) {
    val dT = Position(v1) - Position(v2)
    ...
    Flux(v1) += dT*step
    Flux(v2) -= springForce
    ...
}
WorkgroupLauncher launcher = WorkgroupLauncher_forWorkgroup(001);
ColorBatch colorBatch;
while(launcher.nextBatch(&colorBatch)) {
    Maxforce.ensureSize(colorBatch.kernel_size());
    GlobalContext_copyToGPU();
    for_01<<<batch.blocks(),batch.threads()>>>(
        batch, Force, Position,
        Velocity, Maxforce);
}
```
Results

4 example codes with Liszt and C++ implementations:

- Euler solver from Joe
- Navier-Stokes solver from Joe
- Shallow Water simulator
  - Free-surface simulation on globe as per Drake et al.
  - Second order accurate spatial scheme
- Linear FEM
  - Hexahedral mesh
  - Trilinear basis functions with support at vertices
  - CG solver
Scalar Performance Comparisons

Runtime comparisons between hand-tuned C++ and Liszt
Liszt performance within 12% of C++

<table>
<thead>
<tr>
<th></th>
<th>Euler</th>
<th>Navier-Stokes</th>
<th>FEM</th>
<th>Shallow Water</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh size</td>
<td>367k</td>
<td>668k</td>
<td>216k</td>
<td>327k</td>
</tr>
<tr>
<td>Liszt</td>
<td>0.37s</td>
<td>1.31s</td>
<td>0.22s</td>
<td>3.30s</td>
</tr>
<tr>
<td>C++</td>
<td>0.39s</td>
<td>1.55s</td>
<td>0.19s</td>
<td>3.34s</td>
</tr>
</tbody>
</table>
MPI Performance

4-socket 6-core 2.66Ghz Xeon CPU per node (24 cores), 16GB RAM per node. 256 nodes, 8 cores per node.
GPU Performance

Tesla C2050, Double Precision, compared to single core, Nehalem E5520 2.26Ghz, 8GB RAM
Portability

Tested both pthreads (coloring) and MPI (partitioning) runtime on:

8-core Nehalem E5520 2.26Ghz, 8GB RAM

32-core Nehalem-EX X7560 2.26GHz, 128GB RAM
Takeaways

Bottom-up (from applications) approach worked for us

First attempted to build this as a library

Then invented stencil detection to automate targeting API

Examples are key

Most of the back-end was “extracted” from example codes

Static-Dynamic program analysis split is annoying

Makes code obtuse – lots of generated code for analysis

Makes some things very hard (adaptive meshes…)

Make the compiler part of the runtime? (ArBB…?)

Shared-access machines makes me want to shoot myself in the face