PSAAP II
Particle-laden Turbulence in a Radiation Environment

Liszt 2.0

A DSL for Particles and Meshes Targeted to Exascale Computers

Pat Hanrahan
Alex Aiken
Mattan Erez

Stanford and UT-Austin

Multi-scale, Multi-physics Computers

- Chip (~50 mm [45 nm features]) - teraflop
  - Hybrid chip multiprocessor (CMP)
  - Multi-core (CPU-like) and many-core (GPU-like)
- Board (~50 cm) – fraction of a petaflop
  - Node with multiple CMPs
  - Shared memory multiprocessor (SMP)
- Room of racks (~50 m) - exaflop
  - Cluster of SMPs

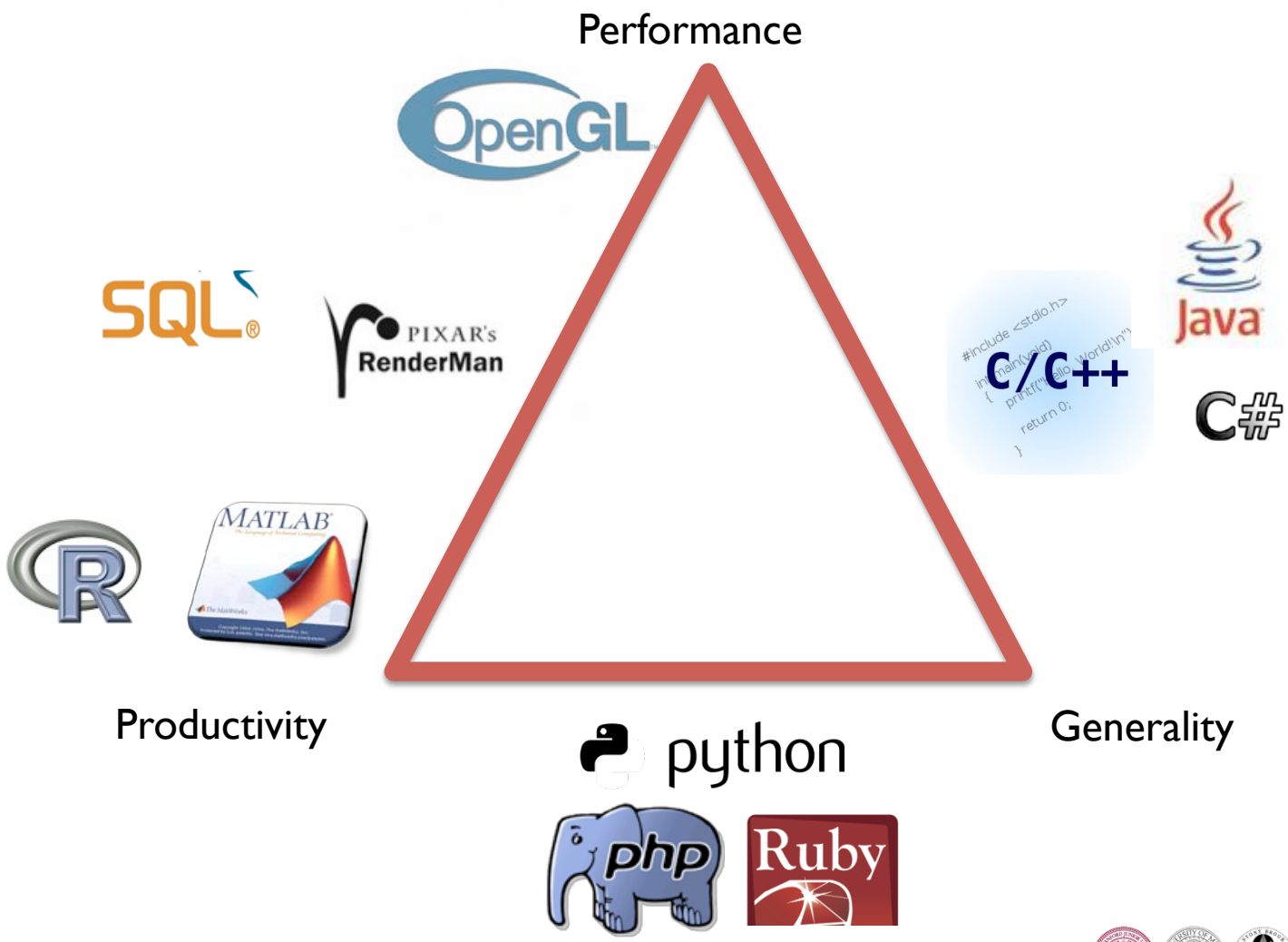
Multiple Software Abstractions

- Cluster
  - Message passing – MPI
- Node
  - Threads and locks – OpenMP, pthreads, …
- CMP
  - GPU cores – OpenCL, CUDA, …

MPI + X + Y

Hard to Program

Portable Parallel Programming
Trade-off Generality vs. Specialization



High-Performance DSL

Our Working Definition: A high-performance DSL is a library/framework that uses domain knowledge to compile optimized code to a parallel platform

NB.

- Not a new language per se; DSL is embedded in a base language
- High-level means that it uses the natural abstractions of the domain

Co-Design

Close knit team

Computation scientists

- *Juan Alonso, Eric Darve, Gianluca Iaccarino, ...*
- Provide application perspective on abstractions
- Provide well-tuned, exemplar system

Computer Scientists

- *Pat Hanrahan, Alex Aiken, Mattan Erez, ...*
- Provide computing perspective on abstractions
- Implement more general system

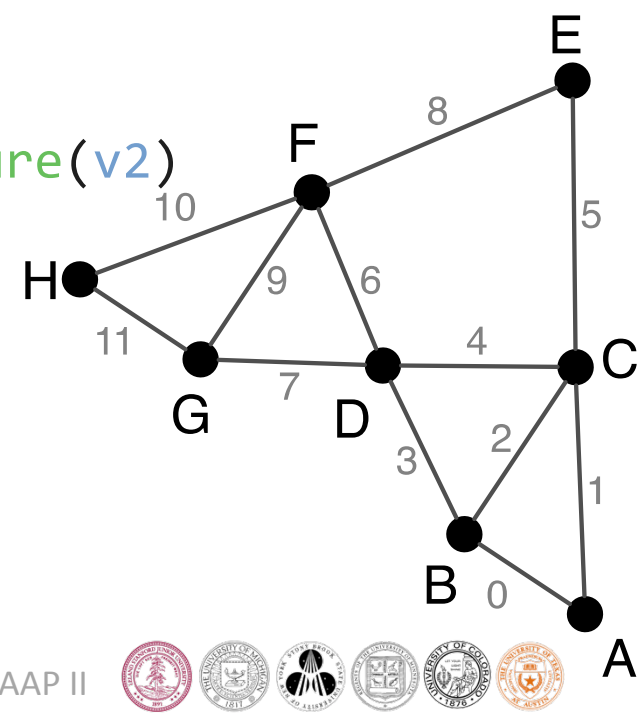
Liszt 1.0

```
var i = 0;
while ( i < 1000 ) {
  Flux(vertices(mesh)) = 0.f;
  JacobiStep(vertices(mesh)) = 0.f;
  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
  }
  i += 1
}
```

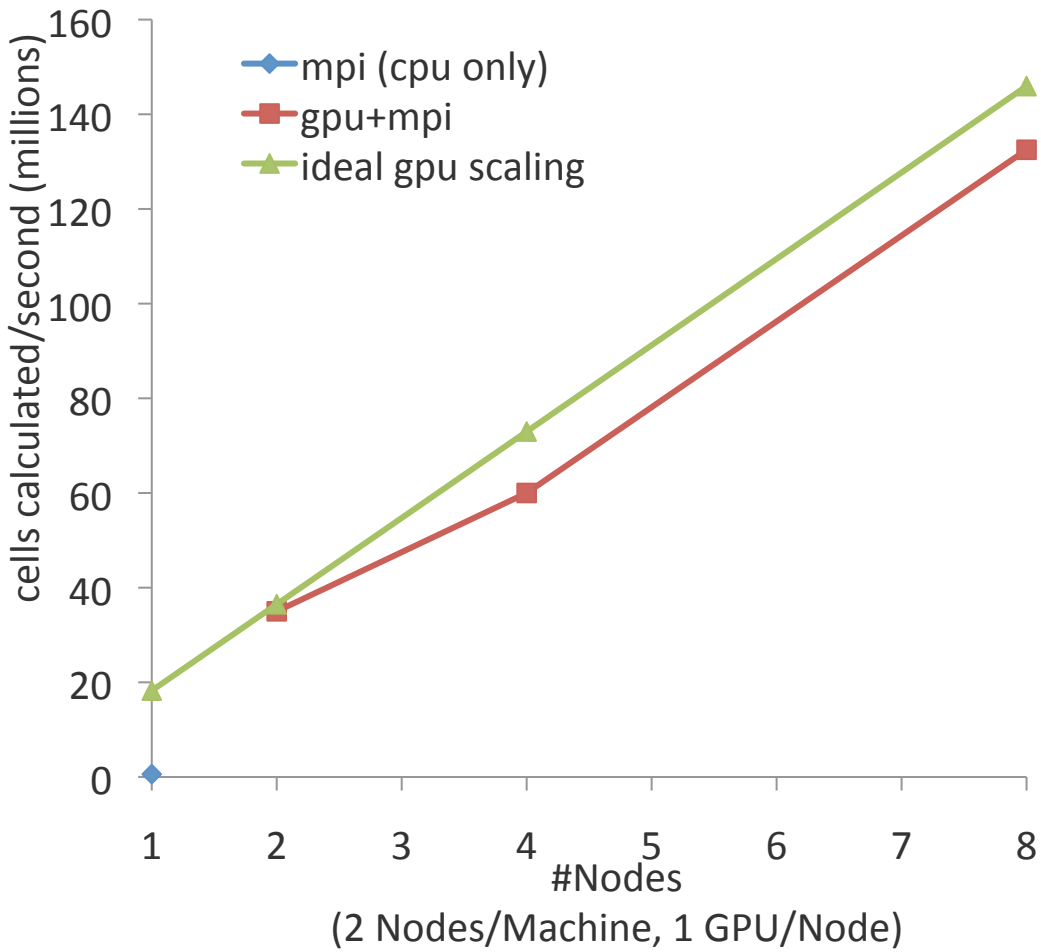
Fields

Mesh (Topology)

Sets



Portable and Performant



If we scale the problem size (double mesh size)
as we scale the cluster size, then we scale

PSAAP: Lessons Learned

- Interoperate, Interoperate, Interoperate
  - e.g. need to call legacy libraries from DSL
  - e.g. need to drop DSL code into apps
  - e.g. multiple interoperating DSLs
- Easy, Easy, Easy
  - Easy to embed DSLs in existing languages
  - Easy to implement domain-specific program transformations
- Dynamic code generation to support auto-tuning and just-in-time compilation, and adaptive meshing

PSAAP2 Proposed Work

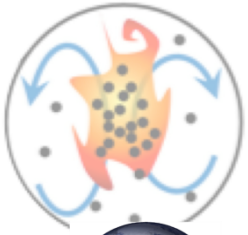
Liszt 2.0 (meshes + particles) - All

Terra (Hanrahan, Aiken)

Legion (Aiken)

Containment domains (Erez)

Terra-Lua System



Terra: Low-level language

- statically typed, monomorphic (like C)
- dynamic compilation via LLVM
- manual memory management

Lua: High-level language

- dynamically typed, polymorphic
- efficient interpreter (LuaJIT)
- garbage collection

Lua&Terra

- similar syntax and shared lexical scope
- co-embedded (can call back and forth)
- state shared through luaJIT FFI

Terra Example

```
--dynamic scripting language (Lua):
function add( a, b )
   return a + b
end
print(add(3,4)) --7

--combined with a low-level language (Terra)
terra addt(a : int, b : int) : int
   return a + b
end
print(addt(3,4)) –7
-- addt is JIT compiled when called
```

Terra Interoperability

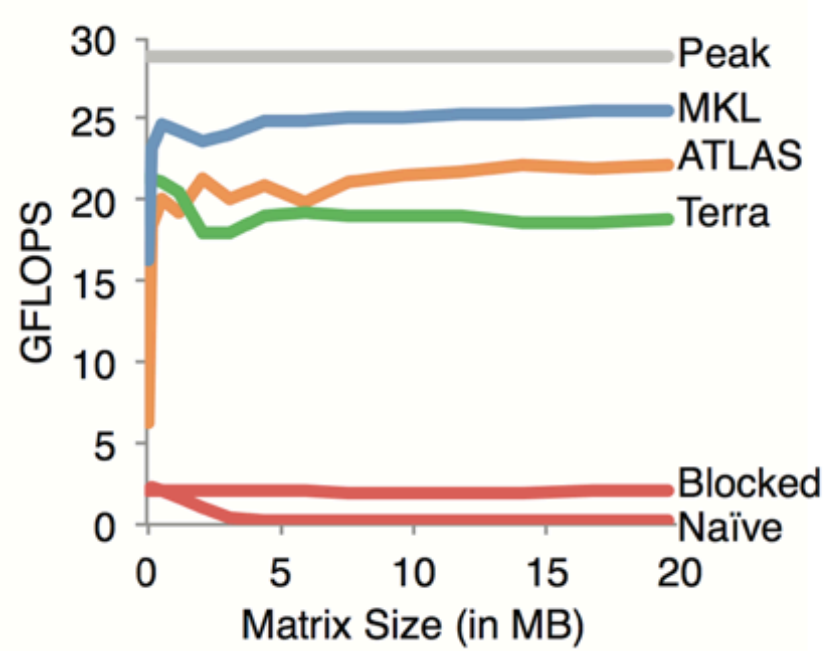
```
local BLAS = terra.includec("gsl/gls_cblas.h")

terra useblas()
  var lda,ldb,ldc = 3,2,2 // : int
  var a,b,c = array(0.11, 0.12, 0.13,
                    0.21, 0.22, 0.23 ), // : double[2][3]
              array(1011, 1012,
                    1021, 1022,
                    1031, 1032 ),
              array(0.00, 0.00,
                    0.00, 0.00 )
    BLAS.sgemm( BLAS.RowMajor,
                BLAS.NoTrans, BLAS.NoTrans,
                2, 2, 3, 1.0, A,lda, B,ldb, 0.0, C,ldc);
end

useblas() — jit and run useblas example
```

Staged Metaprogramming

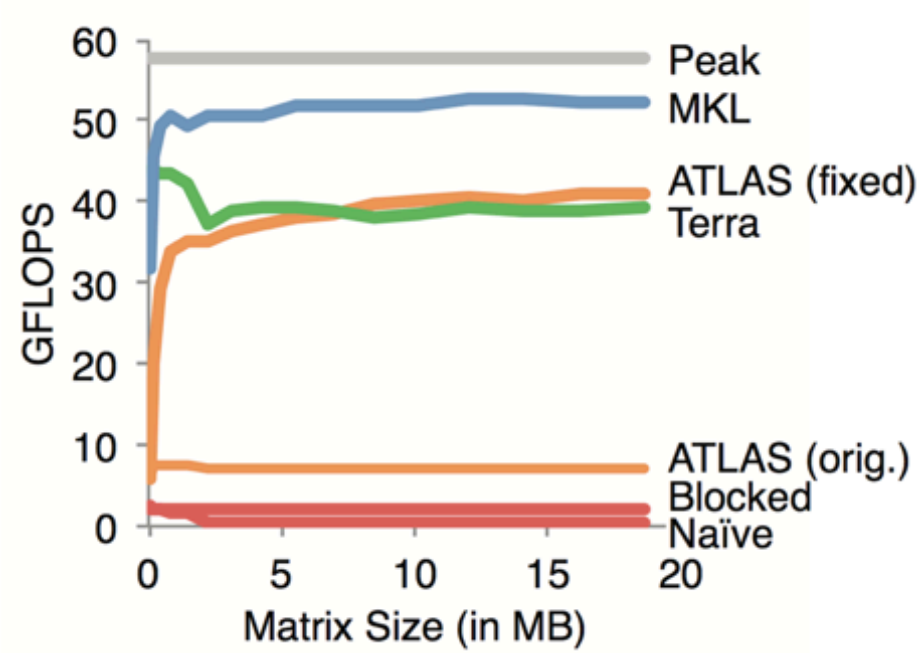
```
function gen_square(x)
   -- backquote ` operation – returns quoted code (AST)
   return `x * x
end

terra mse(a: float, b: float): float
   -- escape operator [ and ] – interpolate lua expression
   return [gen_square(a)] - [gen_square(b)]
end

-- this technique allows us to embed and compile DSLs
-- Inspired by lisp macros, MetaOCaml, MetaHaskell, forth
```

ATLAS in Terra

DGEMM



SGEMM



Optimizations: vectorization, register and cache blocking, unrolling

Size: <150 lines of code (ATLAS much larger and more complex)

Untitled — Edited

Terra

terralang.org

ou're Building a Distributed

Clip to Evernote | San Jose Mercur... | The New York Ti... | CNN | Hack a Day | Google Maps | Wolfram|Alpha: | Facebook | Bookmarks

Terra

Terra

A low-level counterpart to Lua

Download TAR Ball | View On GitHub

Getting Started

API Reference

Publications

Terra is a new low-level system programming language that is designed to interoperate seamlessly with the **Lua** programming language:

```
-- This top-level code is plain Lua code.
print("Hello, Lua!")

-- Terra is backwards compatible with C
-- we'll use C's io library in our example.
C = terralib.includec("stdio.h")

-- The keyword 'terra' introduces
-- a new Terra function.
terra hello(argc : int, argv : &rawstring)
    -- Here we call a C function from Terra
    C.printf("Hello, Terra!\n")
    return 0
end

-- You can call Terra functions directly from Lua
hello(0,nil)

-- Or, you can save them to disk as executables or .o
-- files and link them into existing programs
terralib.saveobj("helloterra",{ main = hello })
```

Like C, Terra is a simple, statically-typed, compiled language with manual memory management. But unlike C, it is designed from the beginning to interoperate with Lua. Terra functions are first-class Lua values created using the `terra` keyword. When needed they are JIT-compiled to machine code.

You can **use** Terra and Lua as…

A scripting-language with high-performance extensions. While the performance of Lua and other dynamic languages is always getting better, a

Zach DeVito
zdevito at stanford dot edu

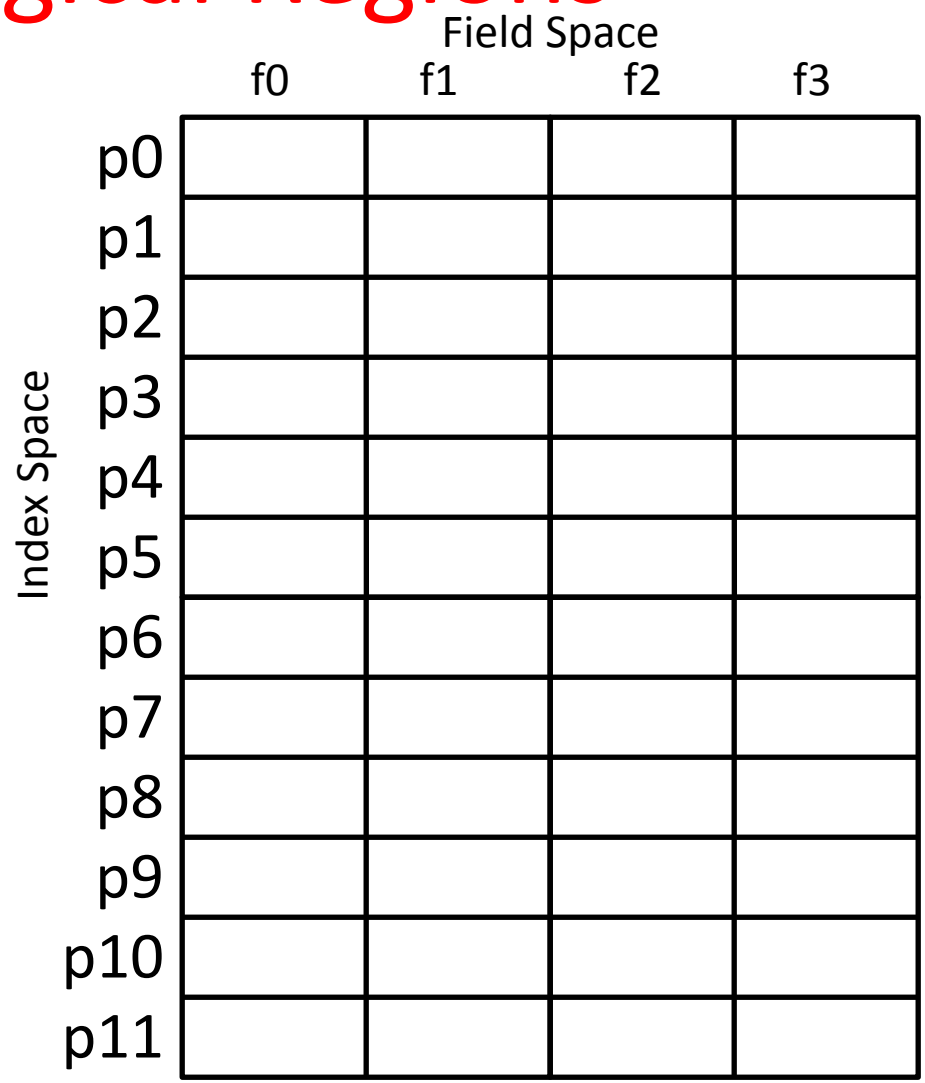
Theme by orderedlist

Legion

- A runtime system for hierarchical, heterogeneous machines
  - Implemented as a C++ library
- Ideas
  - Regions
    - To describe data
  - Deferred execution
    - To deal with high and variable latency
  - Mapping
    - Full programmer control over where data & computation are placed in the machine
- Adding Resilience
  - Maintain information to recover from soft and hard faults

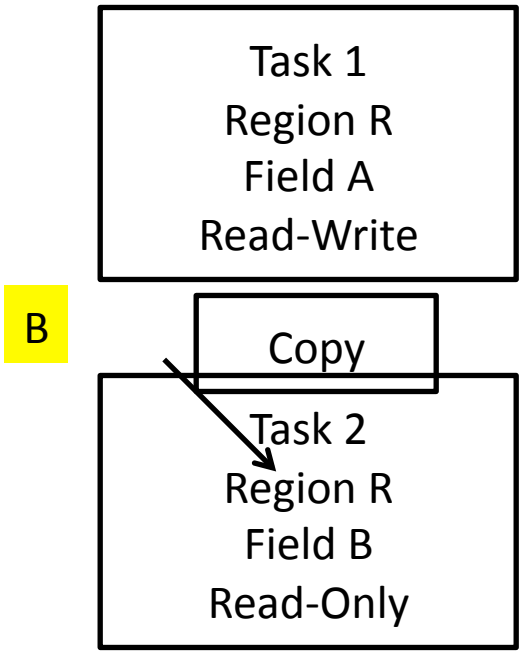
Logical Regions

- Logical regions:
  - Have no implied layout
  - Have no implied location
- Described by:
  - Index space (set of keys)
  - Field space (set of fields)
- Partitioning into subregions
  - In either dimension



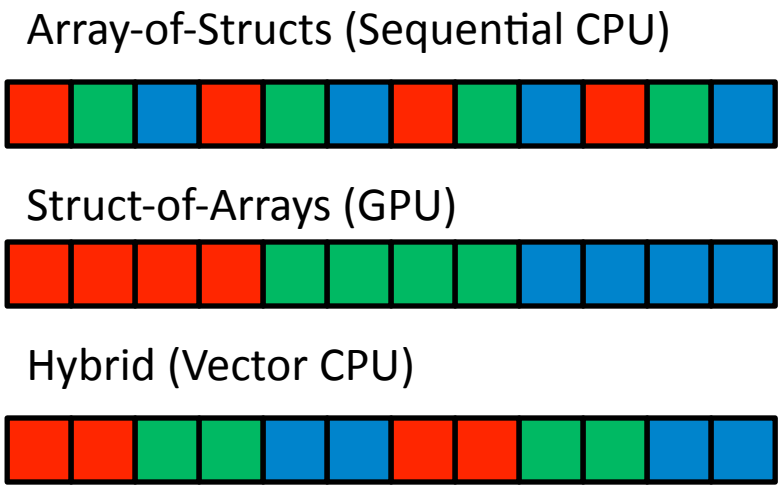
Legion Tasks

- Legion tasks specify:
  - Region usage
  - Field usage
  - Access modes
- Legion runtime:
  - Infers data dependences
  - Inserts copies
- Tasks are asynchronous
  - Deferred execution
  - Hide long latency events



Legion Mapping

- Mapping
  - Assigns tasks to processors
  - Assigns regions to memories



- Programmer selects:
  - Where tasks run
  - Where data is placed
  - Data layout

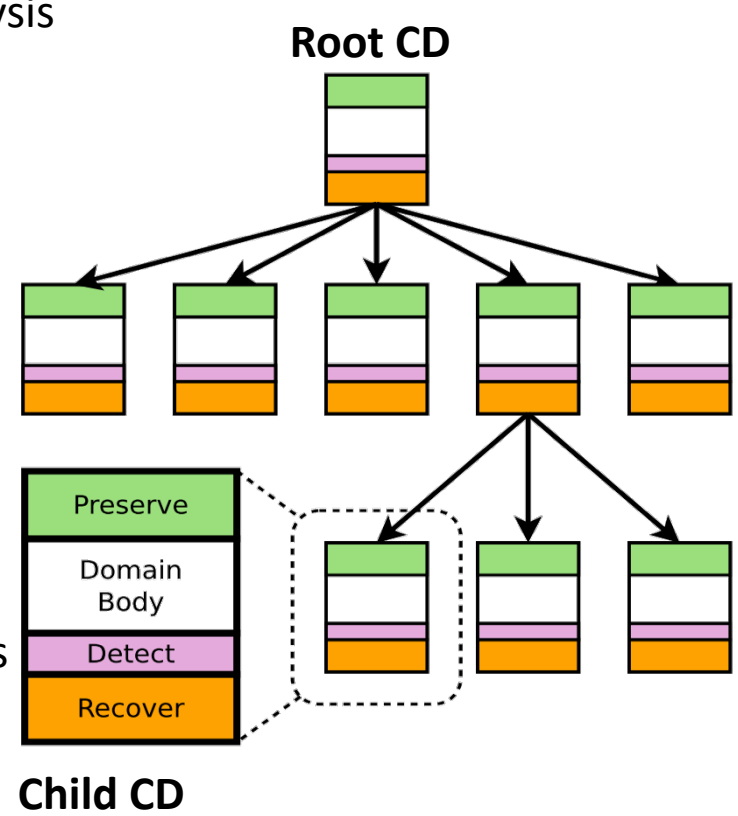
- Provide default mapper

Resilience

- Resilience for exascale computing should be:
  - Hierarchical
    - Preserve data where most efficient and effective
  - Distributed
  - Proportional
    - Tunable redundancy and recovery
    - Different errors/faults handled differently
  - Abstract
    - Portable
    - Amenable to auto-tuning and analysis
- Embed resilience in application

Containment Domains

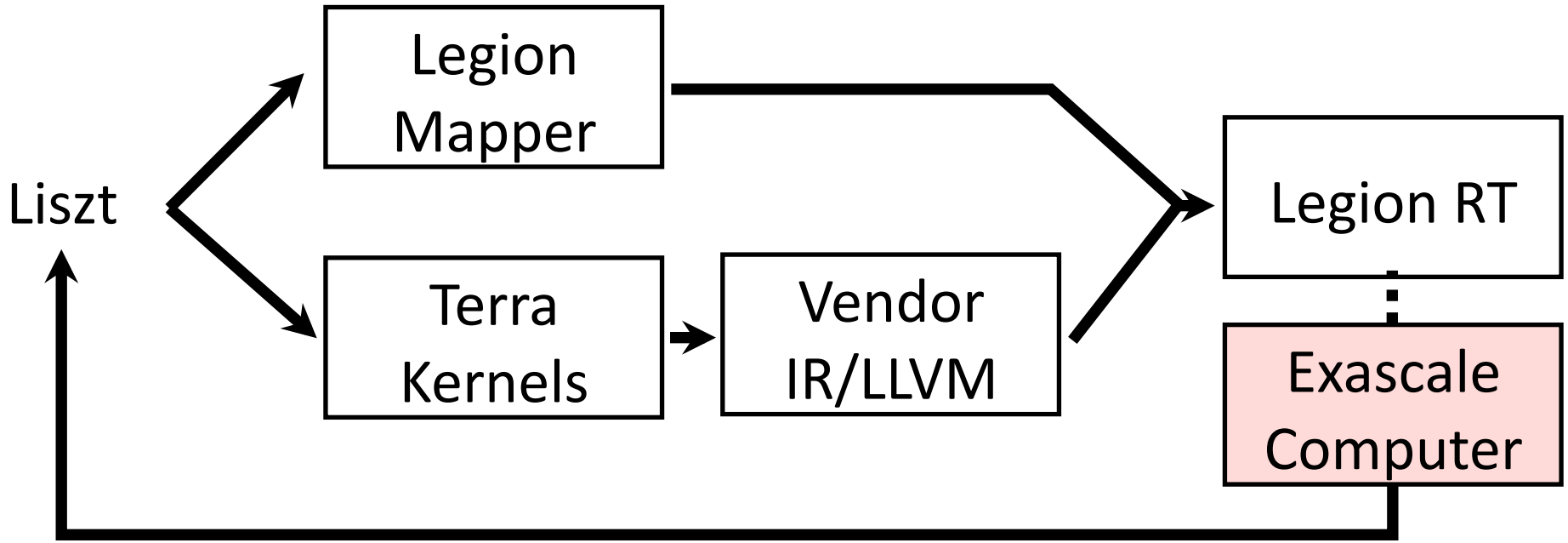
- Single consistent abstraction
  - Encapsulates resilience techniques
  - Spans levels: programming, system, and analysis
- Express resilience as a tree of CDs
  - Match CD, task, and machine hierarchies
  - Escalation for differentiated error handling
- Semantics
  - Erroneous data never communicated
  - Each CD provides recovery mechanism or escalates
- Components of a CD
  - Preserve data for recovery
  - Compute (domain body)
  - Detect failures/errors before domain commits
  - Recover from detected errors



Runtime Research Issues

- Legion interoperation with Lua/Terra
  - Already (mostly) working
- Mapping for Liszt
  - Liszt mapper will control Legion runtime decisions
  - Can we write a general (Liszt) mapper?
- Mapper interactions
  - Some library components (e.g., solvers) may be written in Legion
    - With their own mappers written by library experts
  - How do we compose multiple mappers?
- Containment Domains in Legion
  - Natural fit with Legion's hierarchical partitioning
  - Provide mechanism, expose resilience policy decisions to Liszt

Putting it All Together



Liszt 2.0 in Terra

```
local conduct = liszt_kernel(e in M.edges)
   var v1   = e.head
   var v2   = e.tail
   var dp   = v1.position - v2.position
   var dt   = v1.temperature - v2.temperature
   var step = 1.0 / length(dp)

   v1.flux = v1.flux - dt * step
   v2.flux = v2.flux + dt * step
   v1.jacobistep = v1.jacobistep + step
   v2.jacobistep = v2.jacobistep + step
end
…
for i = 1, 1000 do
   conduct()
   …
end
```

Fields

Mesh (Topology)

Sets

conduct
- can be called from terra
- can be linked to a C library

Meshes + Particles

Extend Liszt abstractions to include particles

- Particle (represent transport and radiation)
- Fields[particle]
  - Position, velocity
  - Mass, energy, albedo, …
- Relationships (modeled with mutable sets)
  - c = cell(p) // return cell containing particle
  - for(val p <- particles(cell)) …
  - for(val p <- neighbors(p)) …

Challenges

Co-optimization

- Optimizations for meshes and particles
- Optimization with conflicting requirements

Particles

- Motion
  - mutable sets
- Preferential concentration
  - Load balancing

Relational Representation

- How to create co-optimized data structures?

- Liszt will accept relations as input
  - Wider range of mesh representations
  - Particle relationships
  - Particle-cell relations

- Inspired by Setl, Loci, OP2 and Alex Aiken's work on compiling data structures from relations
- Inspired by recent work in "NoSQL" databases

Synergistic Efforts
Exascale Co-Design Centers

- ExMatEx, *T. Germann, J. Belak, A. McPherson*
  - Design and implementation of terra
  - Couple Lulesh to CoMD
- ExaCT, *J. Chen, J. Bell, P. McCormick*
  - Design and implementation of legion
  - Orion – regular grids and finite differences
- PSAAP2
  - Liszt 2.0: Meshes + particles
  - Develop code for radiation + particle-laden flow