Liszt

A Domain Specific Language for Mesh-based PDEs on Heterogeneous Parallel Platforms

Z. DeVito, N. Joubert, M. Medina, M. Barrientos, S. Oakley, J. Alonso, E. Darve, F. Ham, P. Hanrahan

ICCS Workshop – January 2011

Multiple Parallel Platforms

Cluster

- Distributed memory over system area network

Many-core GPU (e.g. Cell, Fermi)

- SIMD / SIMT architecture
- Local memory on chip / Separate GPU memory
- Accelerator connected via PCI-E

Multi-core SMP (e.g. 32 core, 4-socket systems)

- Multi-threaded
- Wider vector units
- Complex memory hierarchy, consistency protocols, …

Multiple Parallel Programming Models

Cluster

- MPI

Multi-core SMP (e.g. 32 core, 4-socket systems)

- Threads/locks (pthreads)
- OpenMP
- Thread building blocks, Grand Central
- Transactional memory, …

Many-core GPU (e.g. Cell, Fermi)

- CUDA, OpenCL, Compute Shader
- Ct, Copperhead, …

Complex Heterogeneous Platforms

LANL IBM Roadrunner

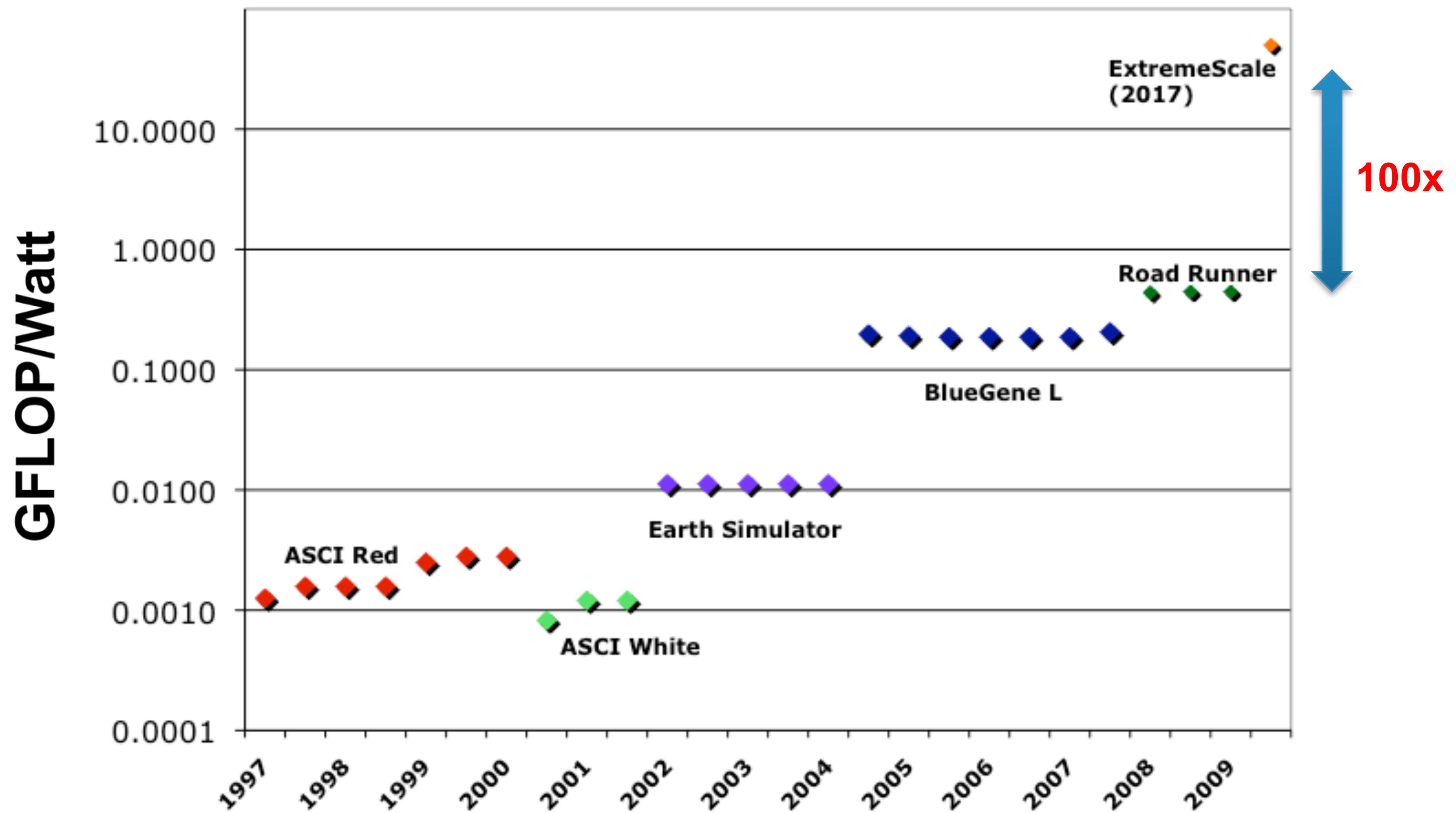
(Opteron + Cell)

ORNL Cray 20 PFLOPs
(Opteron + Fermi)

No agreed upon

programming paradigm

Exascale: 100:1 Improvement Needed



Source: DARPA Exascale Hardware and Software Studies

Should Scientists

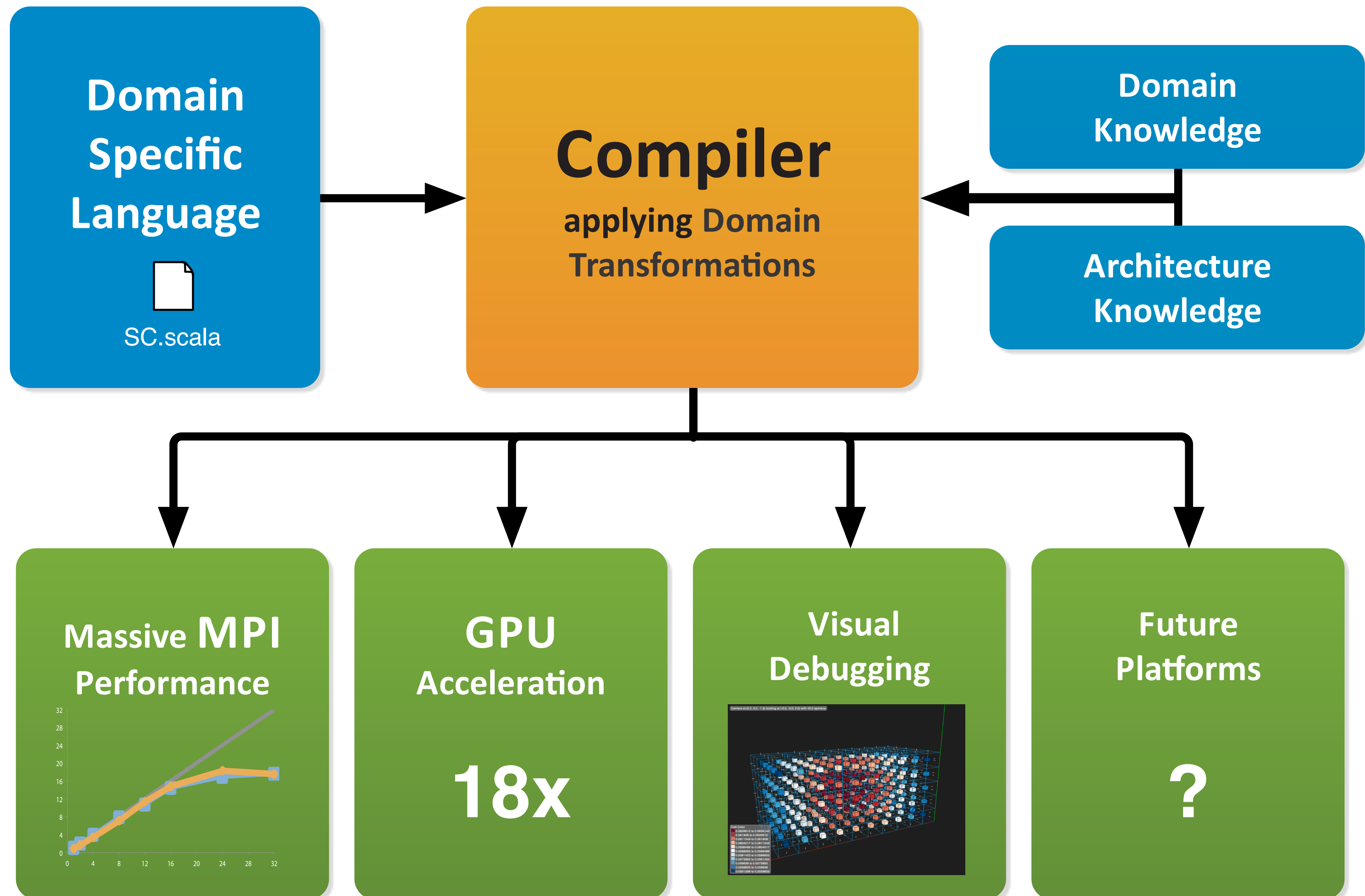
Deal with this Complexity?

Possible Solution:

Domain-Specific

Frameworks and Languages

Liszt DSL: Write one program, Use many architectures



PSAAP's Joe

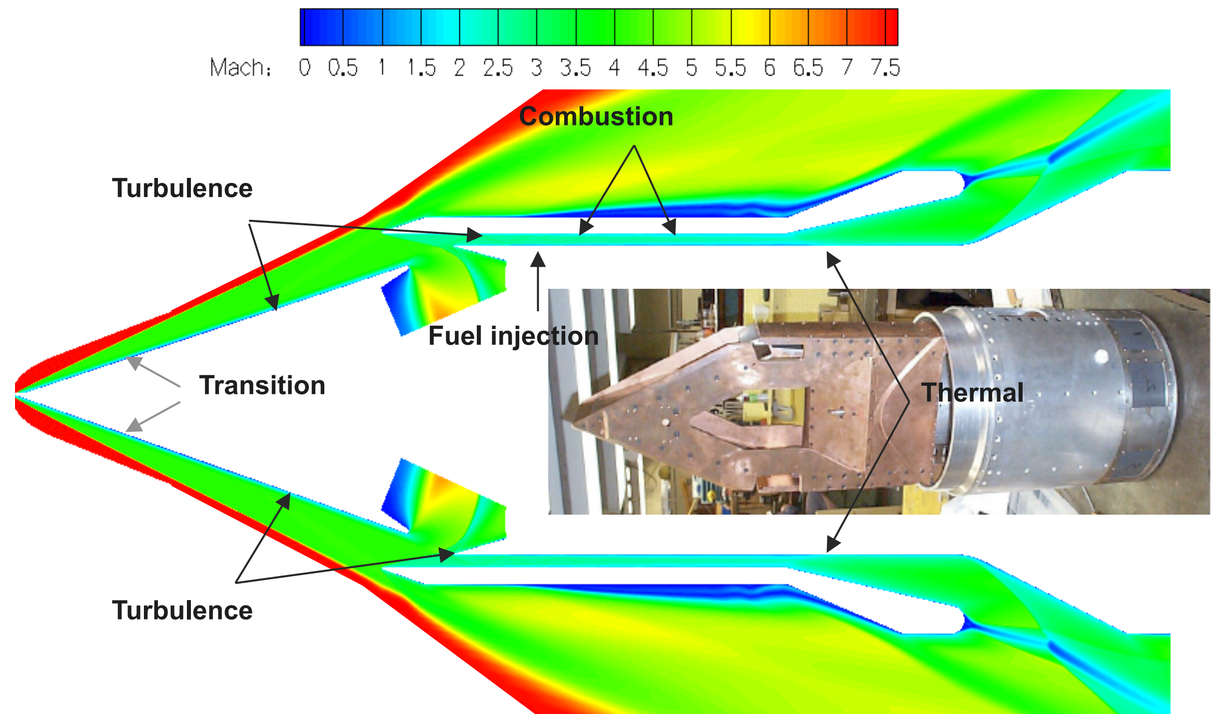
State-of-the-art unstructured Reynolds-averaged Navier Stokes (RANS) solver

Main tool for system-level simulation

- Highly optimized for MPI clusters
- FVM-specific
- Fortran heritage

Common Features

- Large Meshes
- Iterative
- Long Running



Today's Talk:

Language overview

Creating a Domain Specific Language for PDE solving

Language restrictions

Creating an Analyzable Domain Specific Language

Architecture

Building an Analyzable Domain Specific Language

DSL Transformations

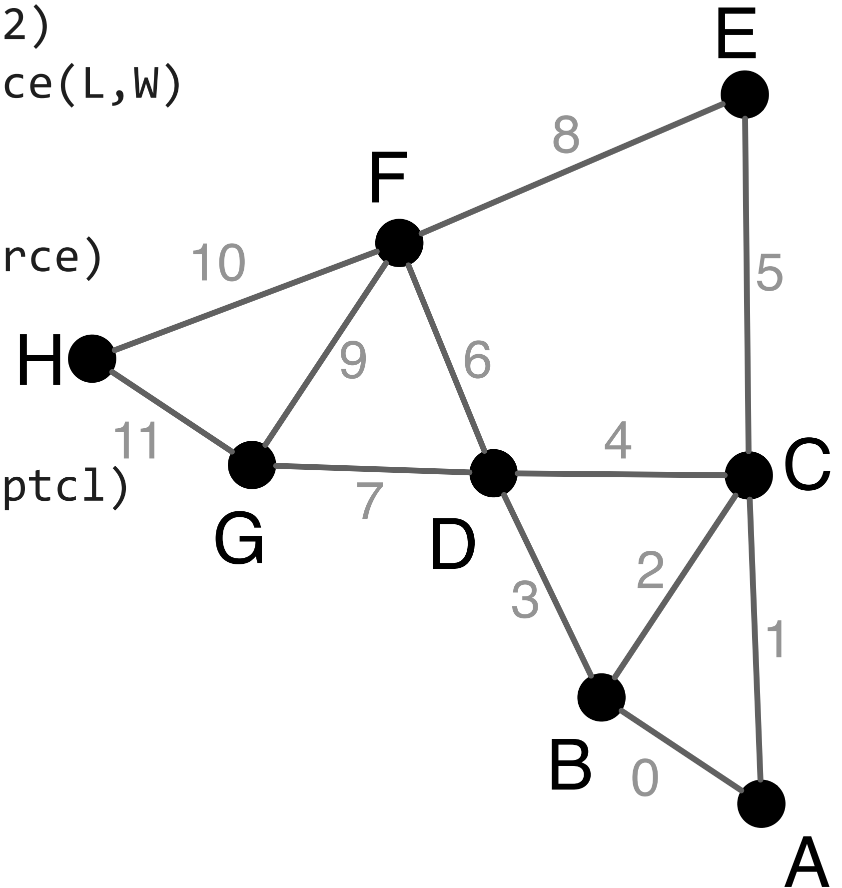
Running an Analyzable DSL on many platforms

Performance Results

Does an Analyzable DSL perform well on all the platforms it supports?

Example: Mass-Spring

```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
deltat = 1 / maxforce*0.5f
for (ptcl <- vertices(mesh)) {
    Velocity(ptcl) += deltat * Force(ptcl)
}
t += deltat
for (ptcl <- vertices(mesh)) {
    Force(ptcl) = Vec(0.f,0.f,0.f)
}
```



Example: Mass-Spring

```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
deltat = 1 / maxforce*0.5f
for (ptcl <- vertices(mesh)) {
    Velocity(ptcl) += deltat * Force(ptcl)
}
t += deltat
for (ptcl <- vertices(mesh)) {
    Force(ptcl) = Vec(0.f,0.f,0.f)
}
```

Mesh Elements

Topology Functions

Fields (Data storage)

Parallelizable for

Designed for Productivity

Built-in mesh interface for arbitrary polyhedra

- `Vertex, Edge, Face, Cell`

Collections of mesh elements

- `Element Sets: faces(cell), edgesCCW(face)`

Mesh-based data storage

- `Fields: val vert_position = position(v)`

Parallelizable iteration

- `For-comprehensions: for( f <- faces(cell) ) { … }`

Parallel Fields Example

A very simple mass-spring forward Euler simulation

C++

```
for (int vi = 0; vi < vertices.size(); vi++) {
    velocity[vi] = deltat * force[vi];
    force[vi] = 0;
}
```

Read velocity and Write velocity

Liszt

```
for (ptcl <- vertices(mesh)) {
    Velocity(ptcl) += deltat * Force(ptcl)
}

for (ptcl <- vertices(mesh)) {
    Force(ptcl) = Vec(0.f,0.f,0.f)
}
```

Read force

Finish for loop

Write force

Restriction: Parallel Field Usage

Fields change state throughout the program – read, write, reduce. A field **cannot** change state during a for comprehension.

- Parallel `for` loops:
  - Liszt prevents creating an undefined value in a field by limiting the type of access per loop.
- Allow platform specific actions to produce correct final values
  - Deferred computation & communication
  - Overlap communication with computation

Restriction: Static Mesh Variables

We only allow single static assignment of mesh variables.

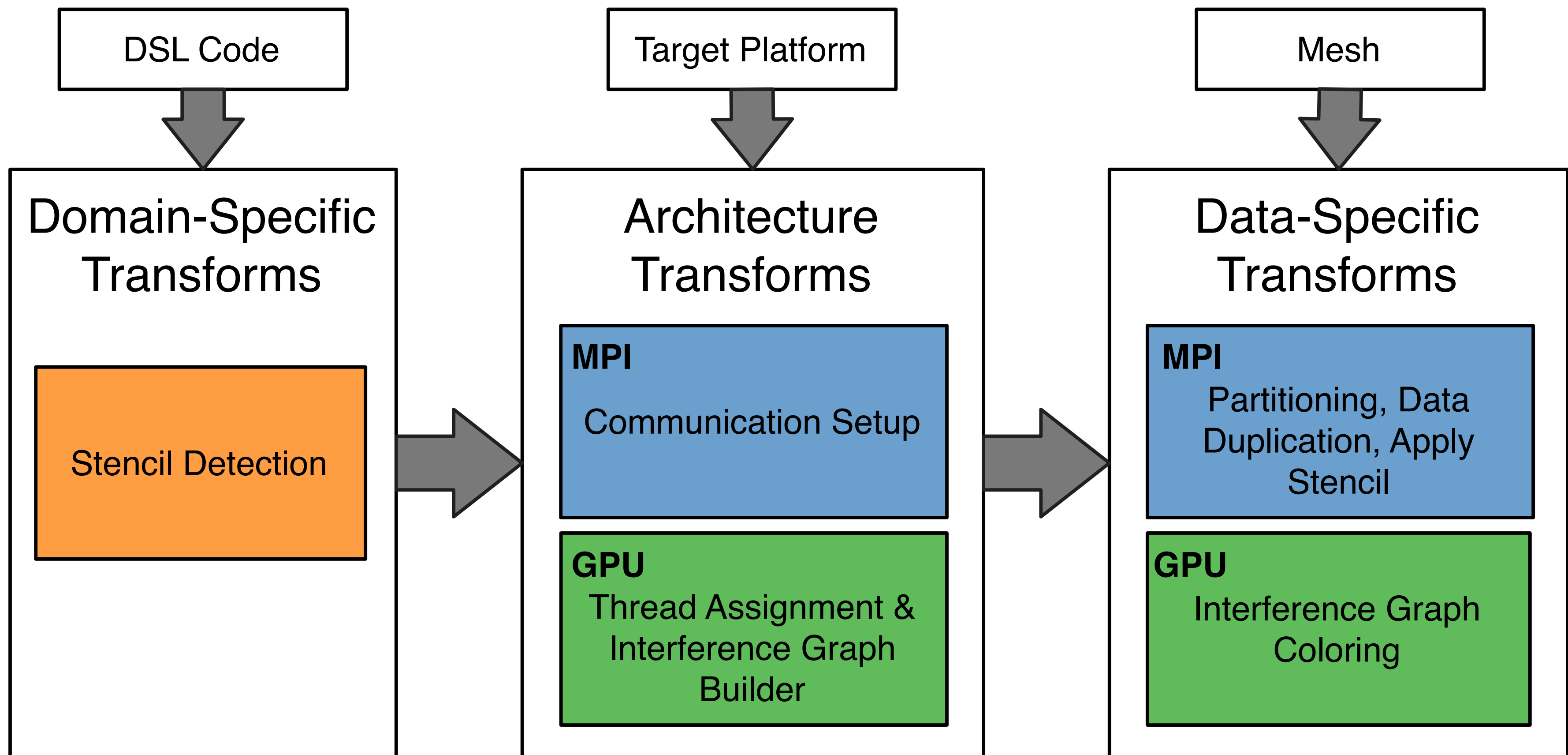
- Code cannot access mesh topology arbitrarily far long the mesh without explicitly accessing it.
- We can analyze code to exactly know what mesh elements it uses.
- No recursion on mesh variables

Compiler Error:

```
for (f <- faces(mesh)) {
    var t = 0
    var e = edge(f,0)
    while (t < 10) {
        e = edgeAroundFace(f,e)
        t += 1
    }
}
```

Only allow Single Static Assignment of mesh topology variables

Architecture

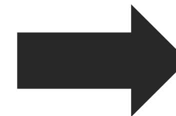


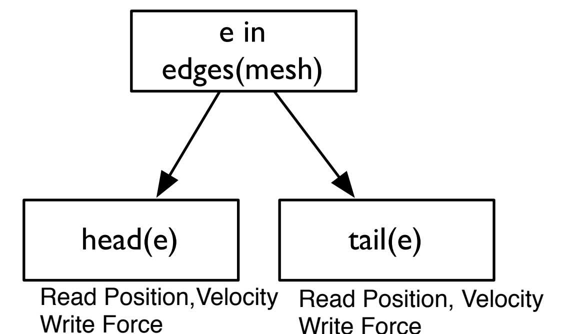
Domain Specific Transform:
Stencil Detection

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

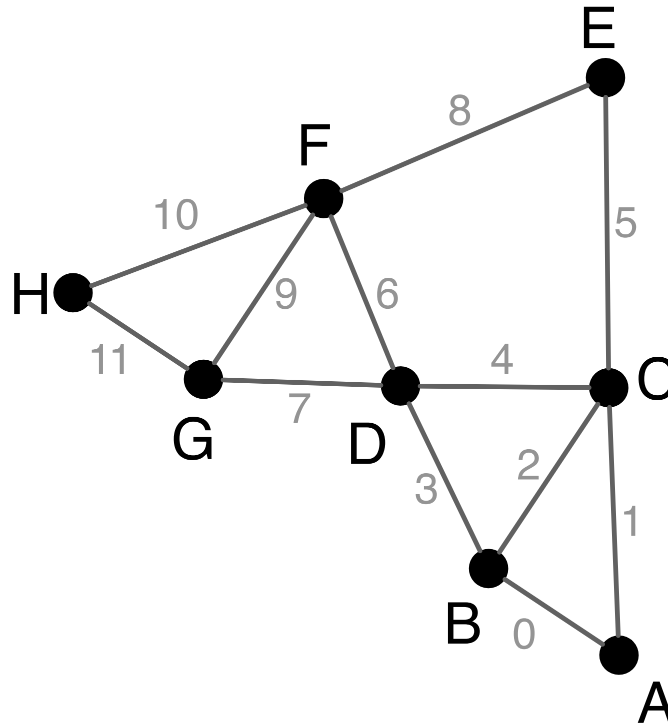
```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
```





Domain Specific Transform: Stencil Detection

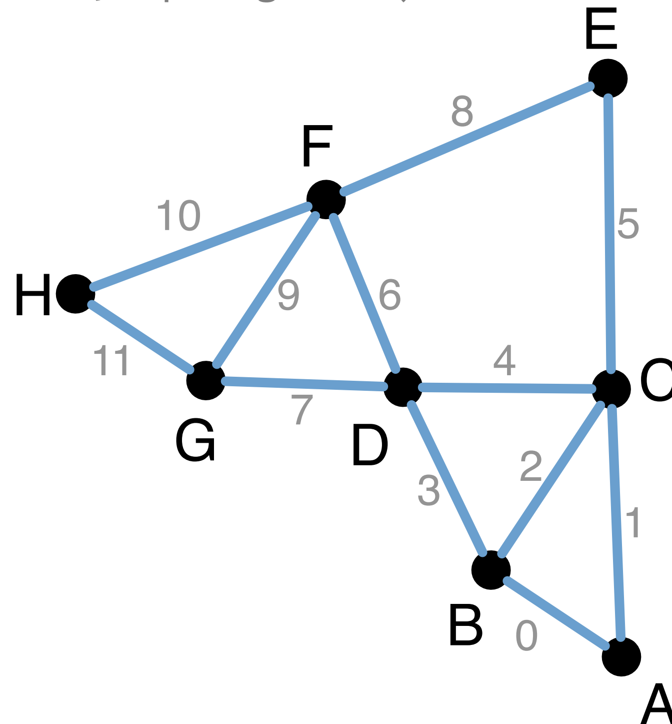
```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
...
```



Domain Specific Transform:
Stencil Detection

```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
...
```

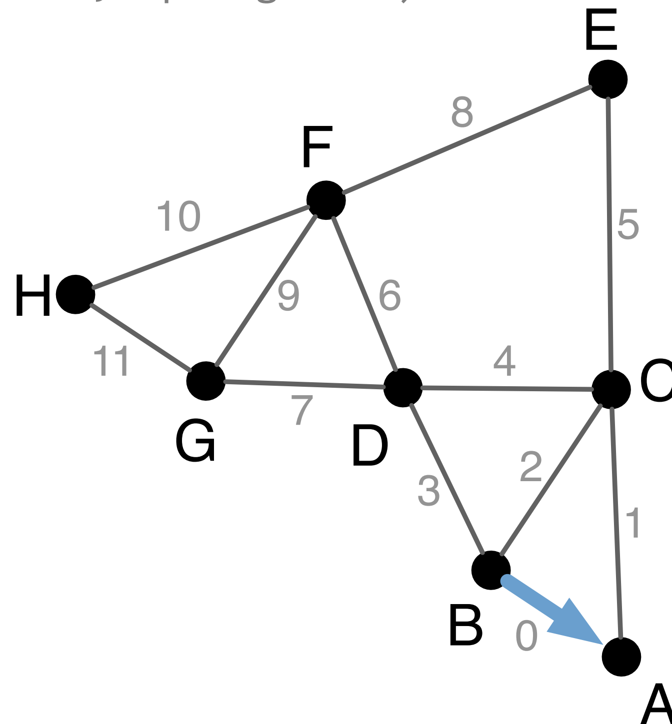
e in
edges(mesh)



Domain Specific Transform:
Stencil Detection

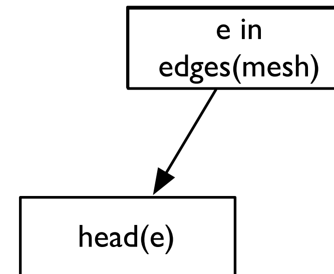
```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
...
```

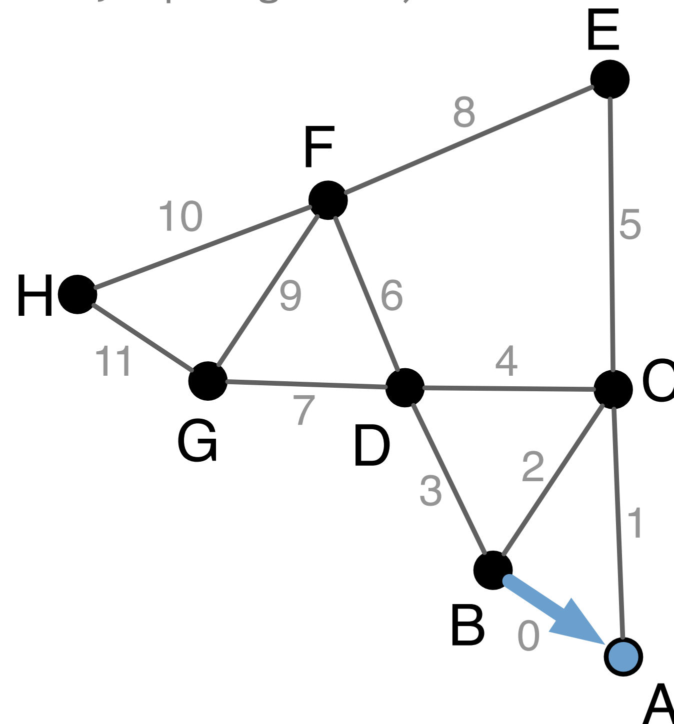
e in edges(mesh)



Domain Specific Transform: Stencil Detection

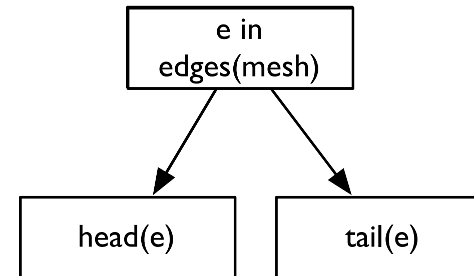
```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
...
```

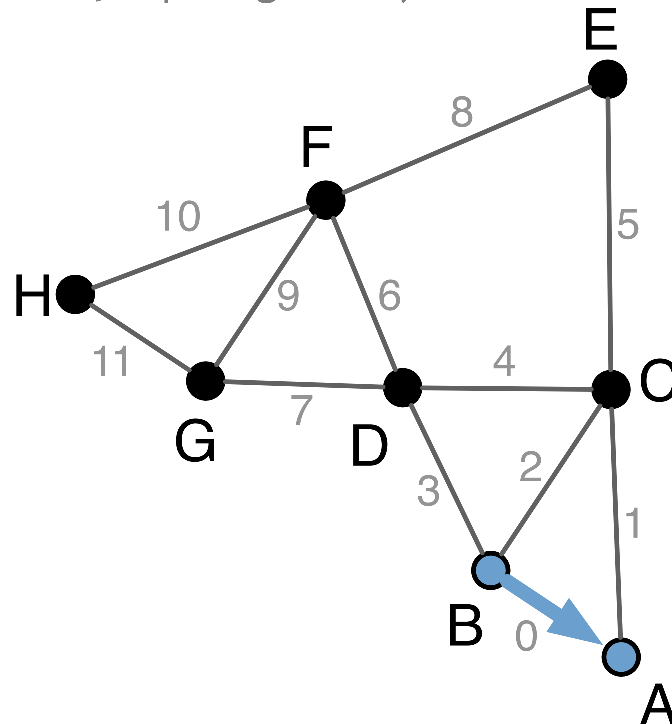




Domain Specific Transform:
Stencil Detection

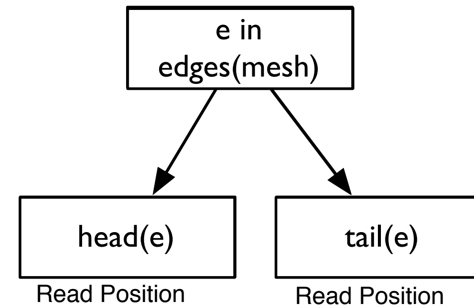
```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
...
```

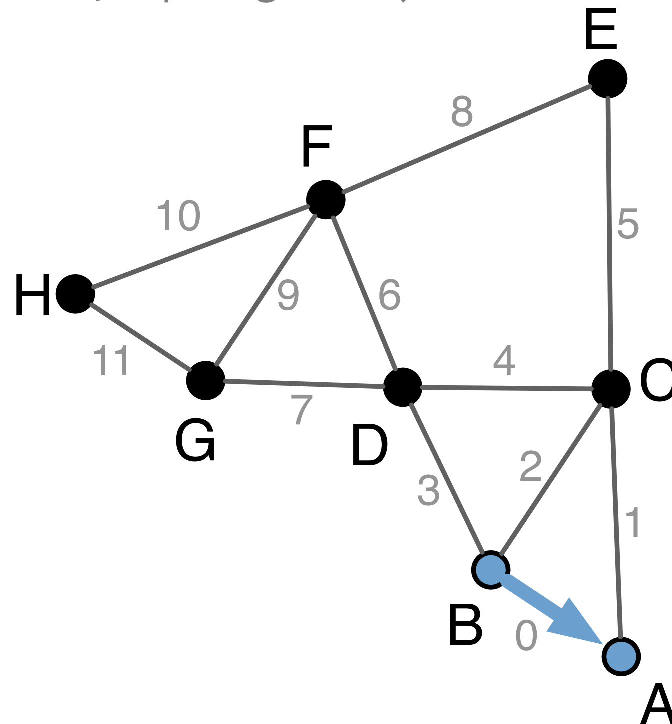




Domain Specific Transform:
Stencil Detection

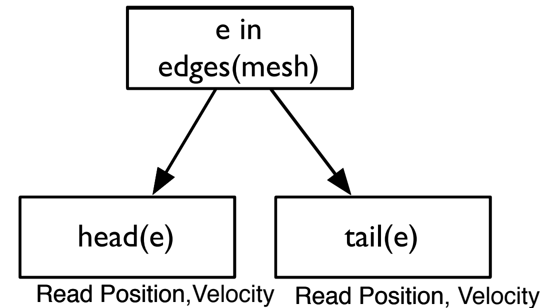
```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
...
```

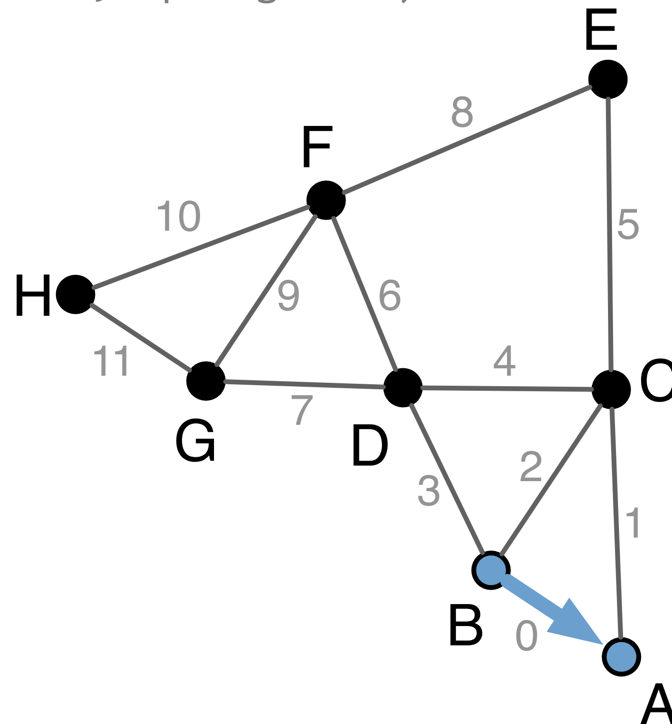




Domain Specific Transform: Stencil Detection

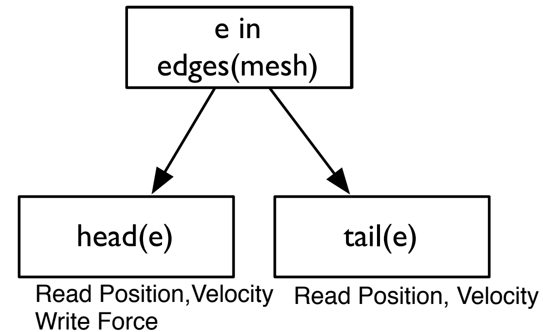
```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
...
```

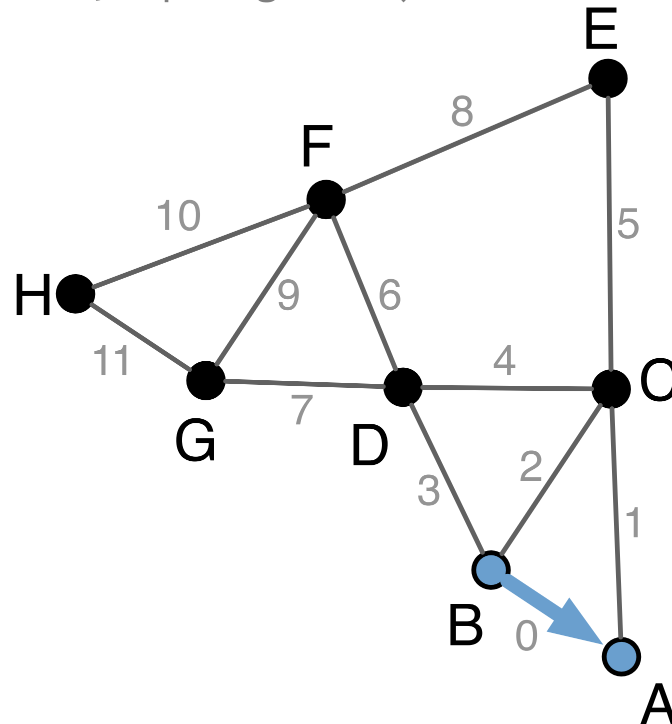




Domain Specific Transform:
Stencil Detection

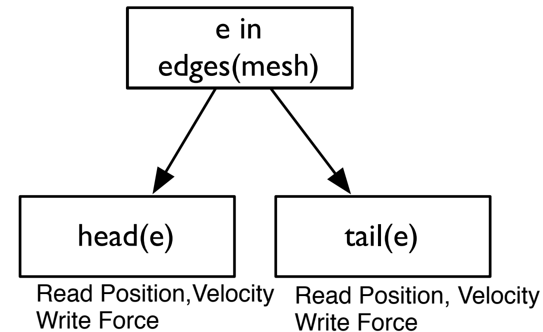
```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
...
```

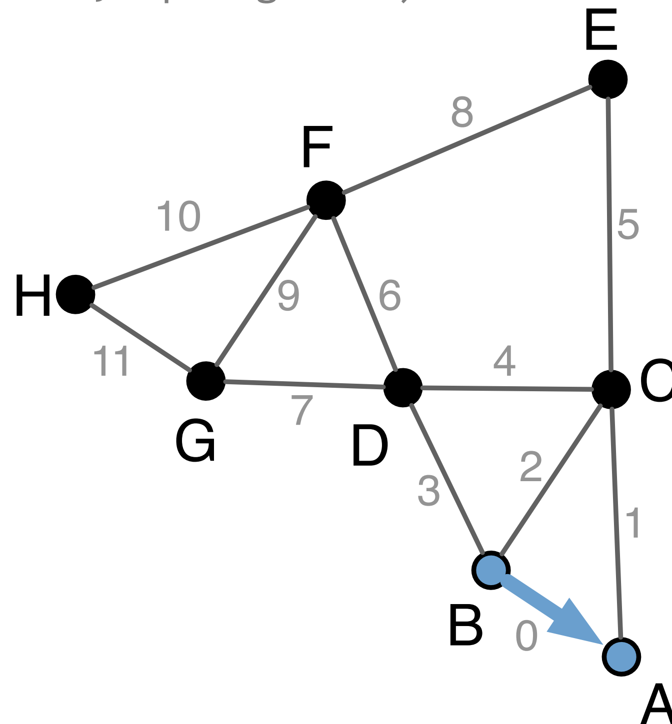




Domain Specific Transform: Stencil Detection

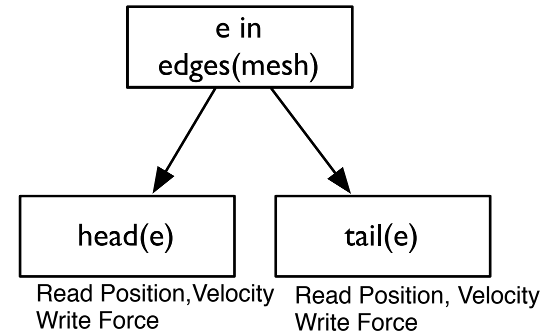
```
for (spring <- edges(mesh)) {
    val v1 = head(spring)
    val v2 = tail(spring)
    val L = Position(v1) - Position(v2)
    val W = Velocity(v1) - Velocity(v2)
    val springForce = dampedSpringForce(L,W)
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce = max(maxforce, springForce)
}
...
```

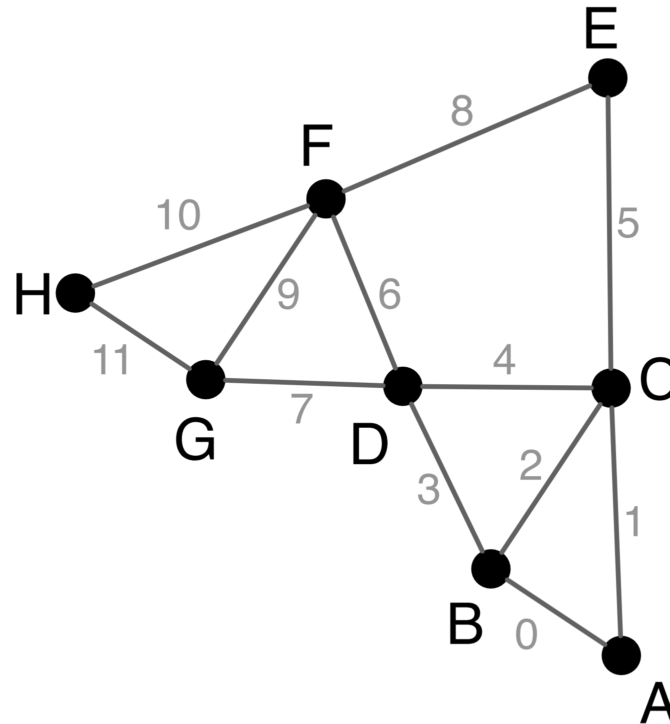




Domain Specific Transform:
Stencil Detection

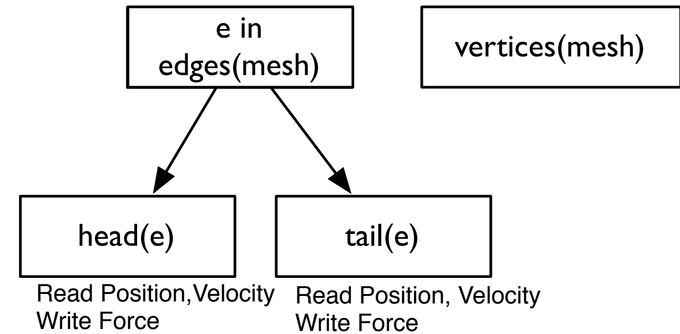
```
deltat = 1 / maxforce*0.5f
for (ptcl <- vertices(mesh)) {
    Velocity(ptcl) += deltat * Force(ptcl)
}
t += deltat
for (ptcl <- vertices(mesh)) {
    Force(ptcl) = Vec(0.f,0.f,0.f)
}
```

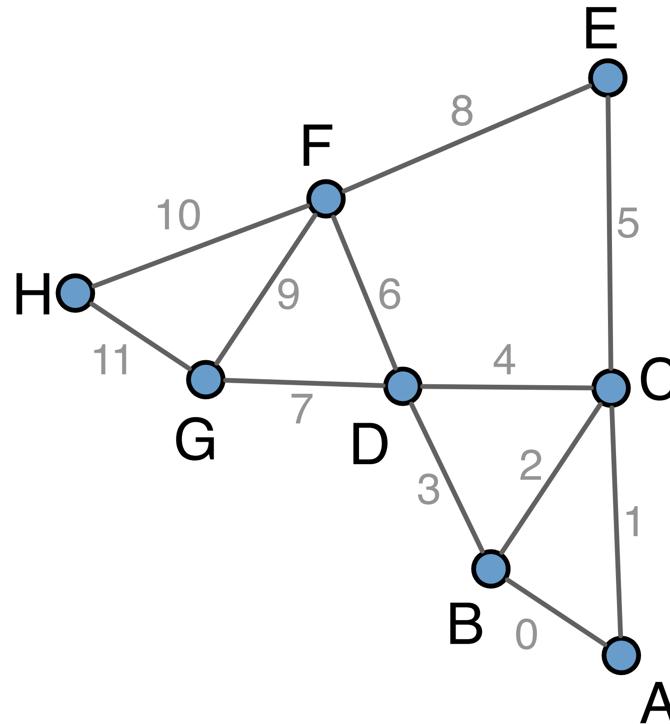




Domain Specific Transform:
Stencil Detection

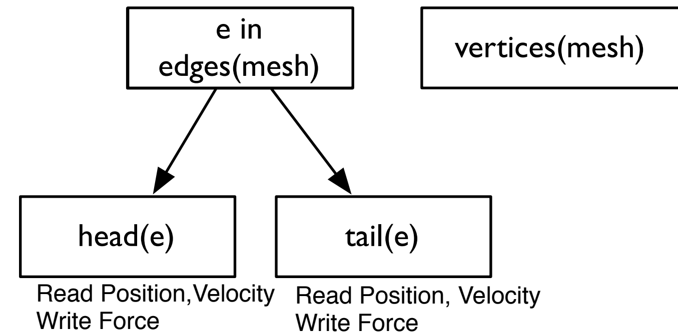
```
deltat = 1 / maxforce*0.5f
for (ptcl <- vertices(mesh)) {
    Velocity(ptcl) += deltat * Force(ptcl)
}
t += deltat
for (ptcl <- vertices(mesh)) {
    Force(ptcl) = Vec(0.f,0.f,0.f)
}
```

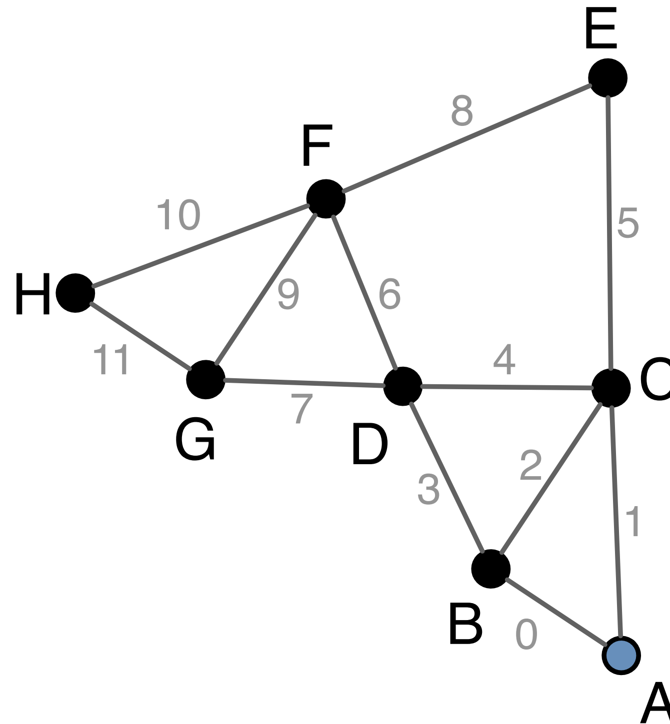




Domain Specific Transform:
Stencil Detection

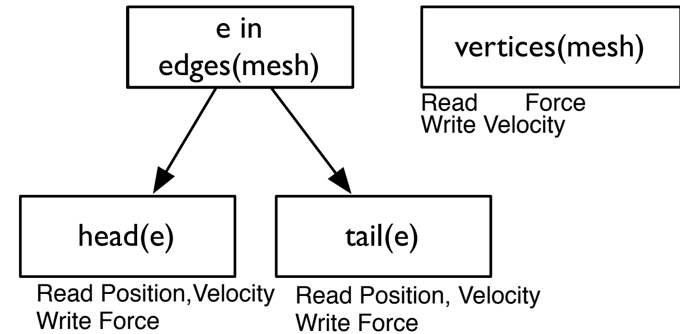
```
deltat = 1 / maxforce*0.5f
for (ptcl <- vertices(mesh)) {
    Velocity(ptcl) += deltat * Force(ptcl)
}
t += deltat
for (ptcl <- vertices(mesh)) {
    Force(ptcl) = Vec(0.f,0.f,0.f)
}
```

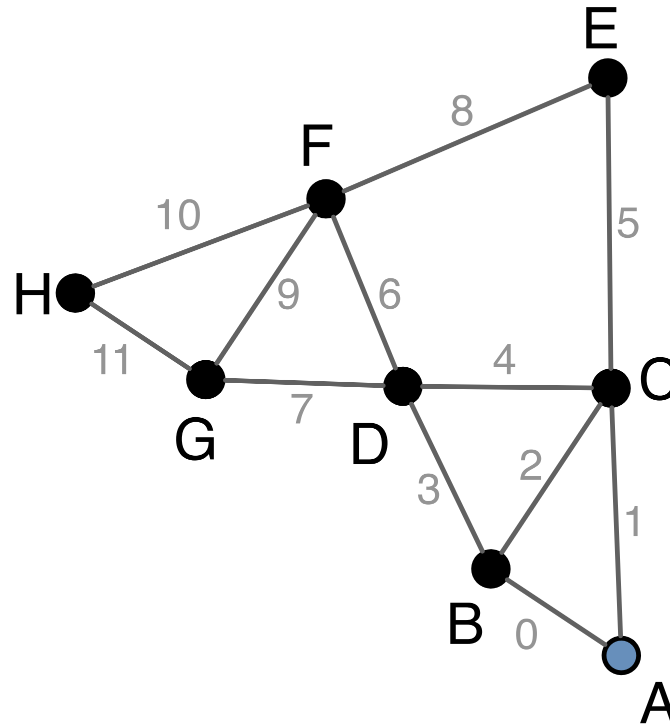




Domain Specific Transform: Stencil Detection

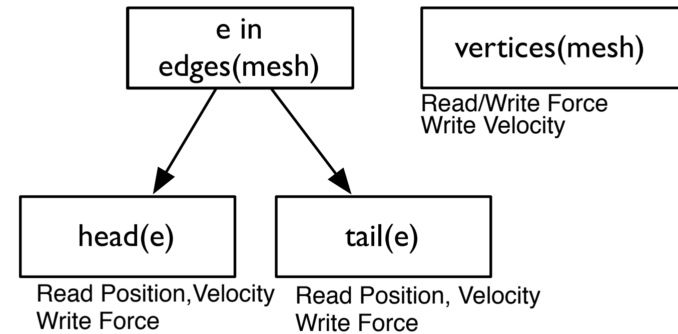
```
deltat = 1 / maxforce*0.5f
for (ptcl <- vertices(mesh)) {
    Velocity(ptcl) += deltat * Force(ptcl)
}
t += deltat
for (ptcl <- vertices(mesh)) {
    Force(ptcl) = Vec(0.f,0.f,0.f)
}
```

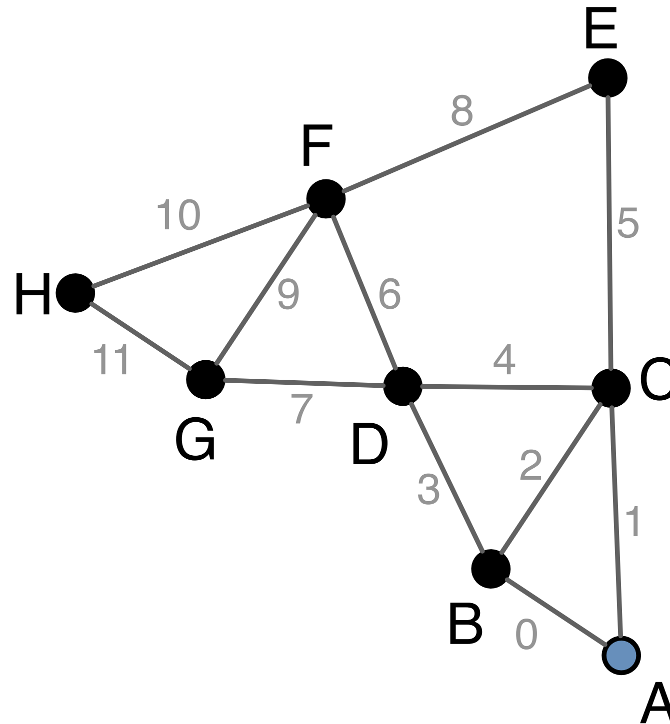




Domain Specific Transform:
Stencil Detection

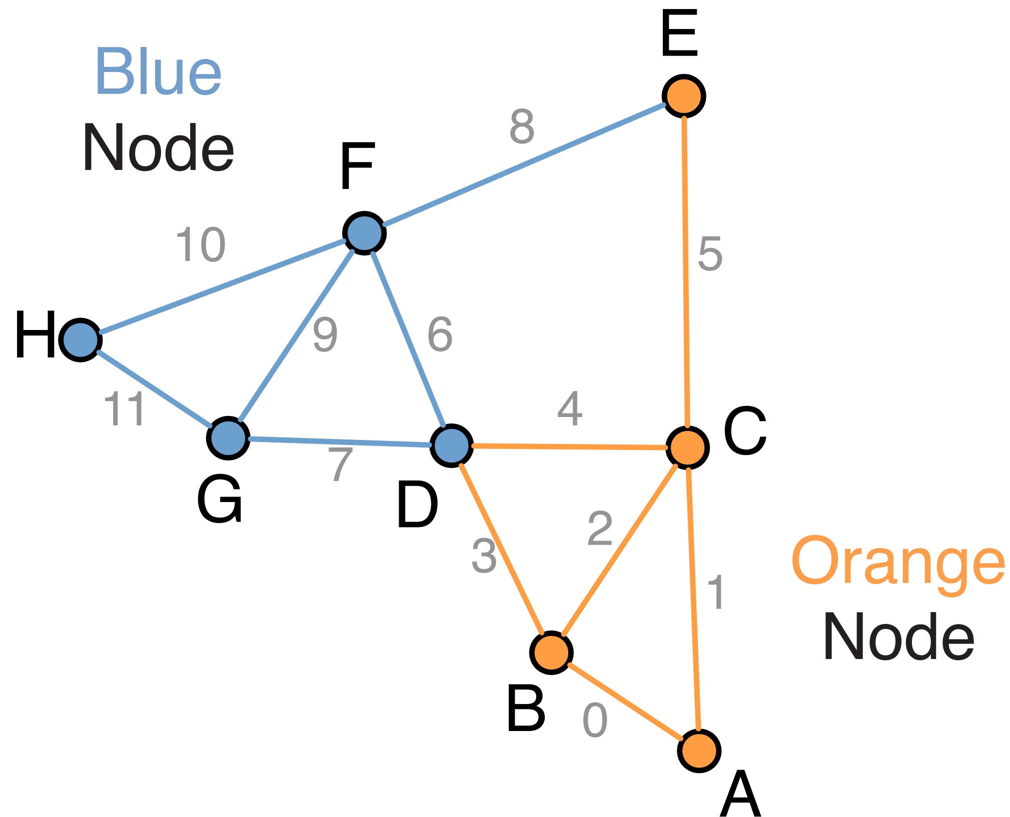
```
deltat = 1 / maxforce*0.5f
for (ptcl <- vertices(mesh)) {
    Velocity(ptcl) += deltat * Force(ptcl)
}
t += deltat
for (ptcl <- vertices(mesh)) {
    Force(ptcl) = Vec(0.f,0.f,0.f)
}
```





MPI: Partitioning with Ghosts

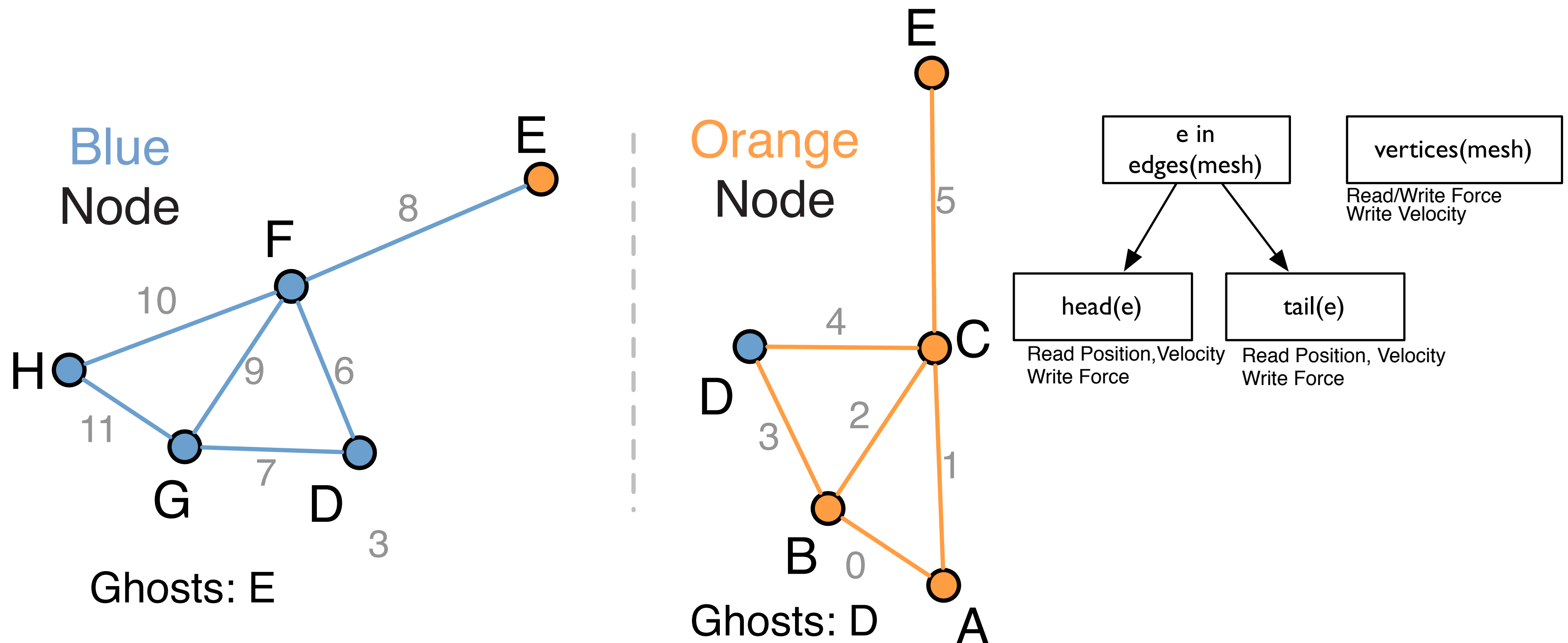
1. Partition Mesh (ParMETIS, G. Karypis)



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



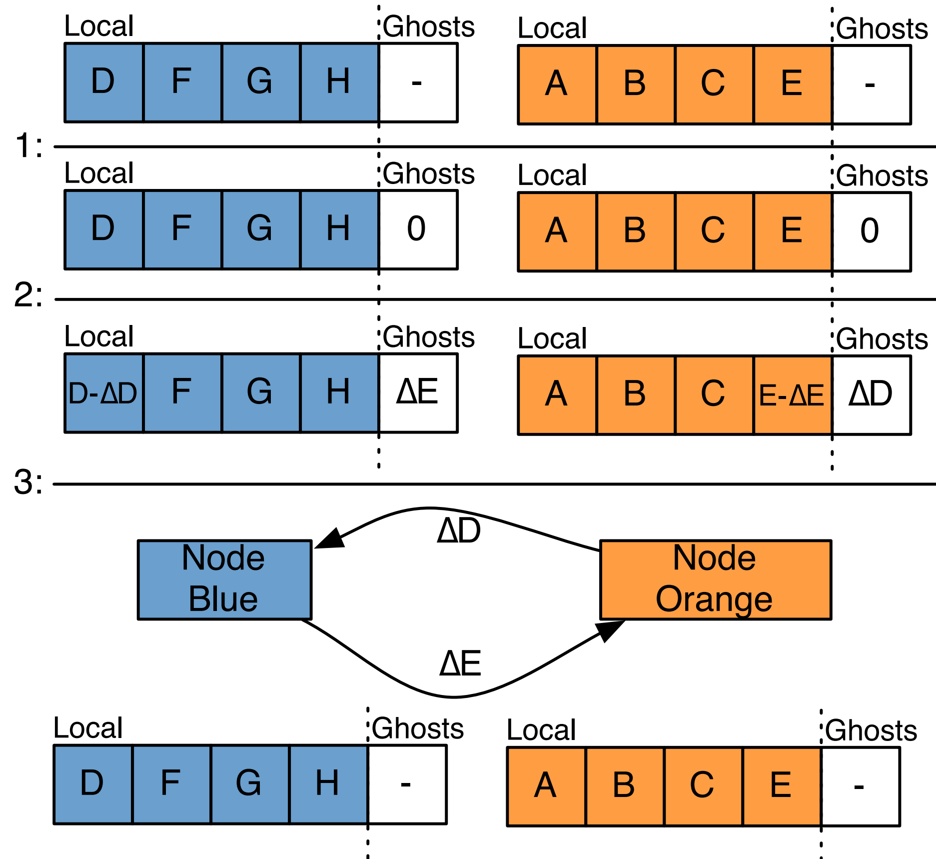
MPI: Partitioning with Ghosts

3. Annotate for-comprehensions with field preparation statements

```
Force.ensureState<LISZT_SUM>();
Position.ensureState<LISZT_READ>();
Velocity.ensureState<LISZT_READ>();
Maxforce.ensureState<LISZT_MAX>();
for (spring <- edges(mesh)) {
    val L = Position(v1) - Position(v2)
    ...
    Force(v1) += springForce
    maxforce = max(maxforce, springForce)
}
Velocity.ensureState<LISZT_SUM>();
Force.ensureState<LISZT_READ>();
for (ptcl <- vertices(mesh)) {
    Velocity(ptcl) += deltat * Force(ptcl)
}
Force.ensureState<LISZT_ASSIGN>();
for (ptcl <- vertices(mesh)) { Force(ptcl) = ...}
```

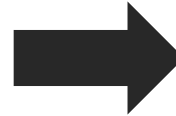
MPI: Partitioning with Ghosts

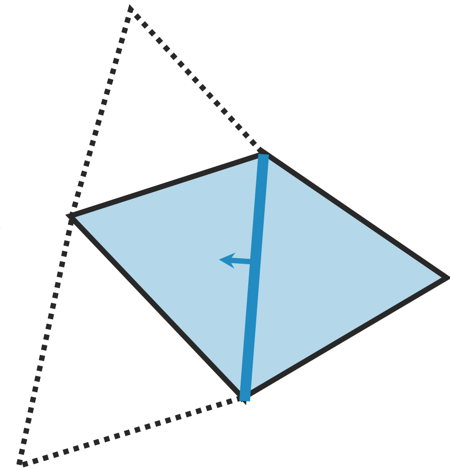
4. MPI communication is batched during for-comprehensions and only transferred when necessary



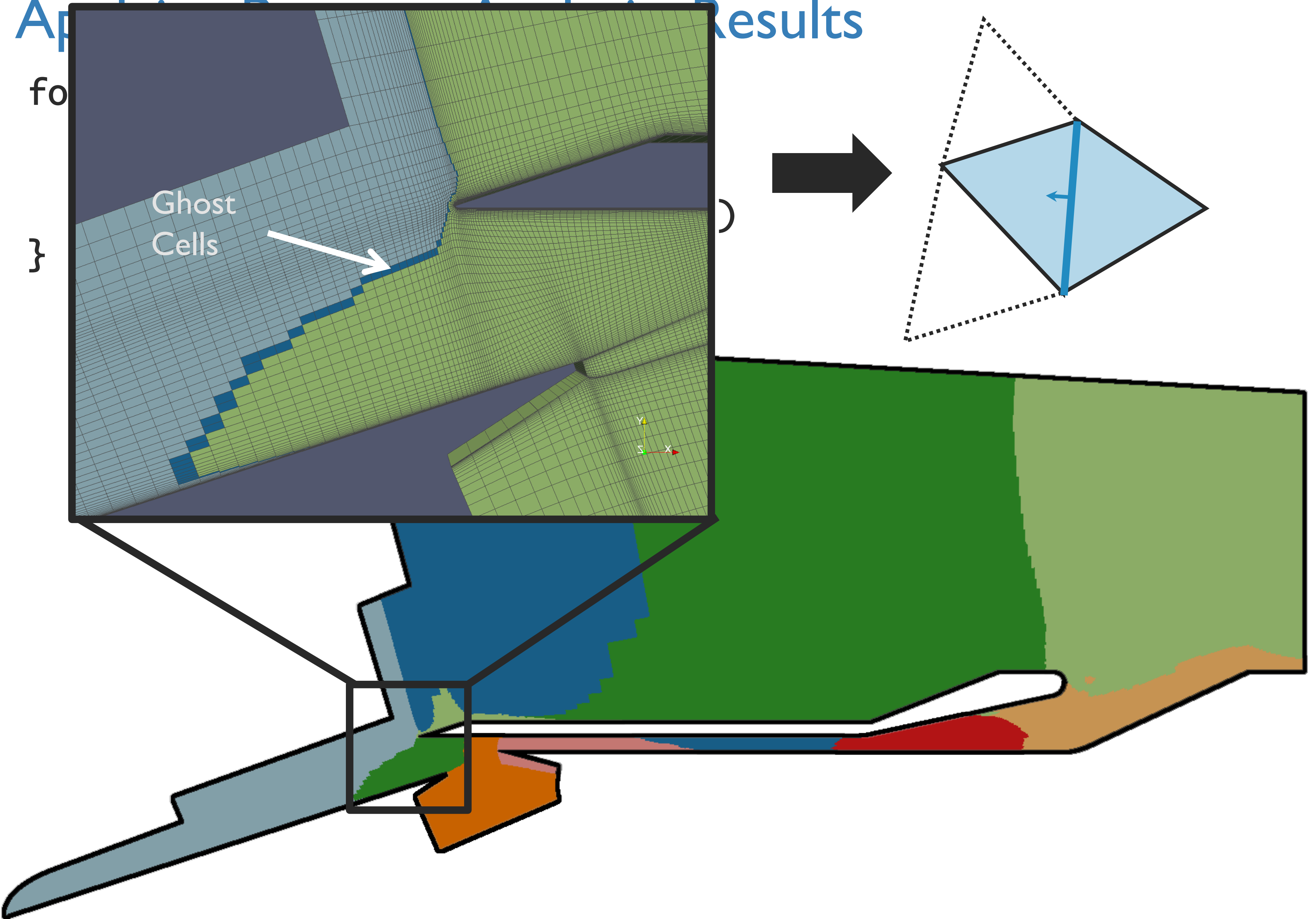
Applying Program Analysis: Results

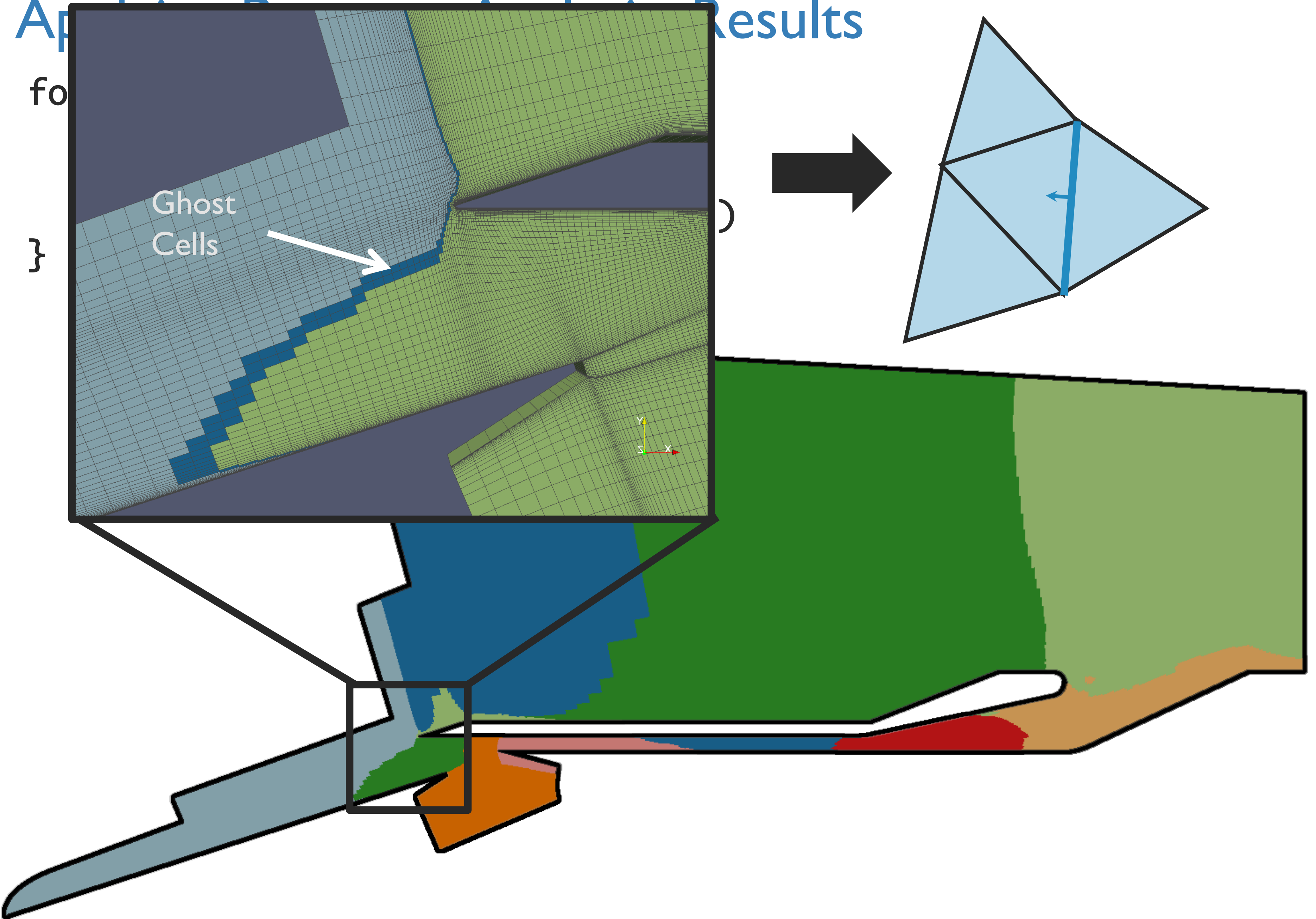
```
for(f <- faces(mesh)) {
  rhoOutside(f) :=
    calc_flux( f,rho(outside(f) ))
    + calc_flux( f,rho(inside(f) ))
}
```





Initial Partition
(ParMETIS)

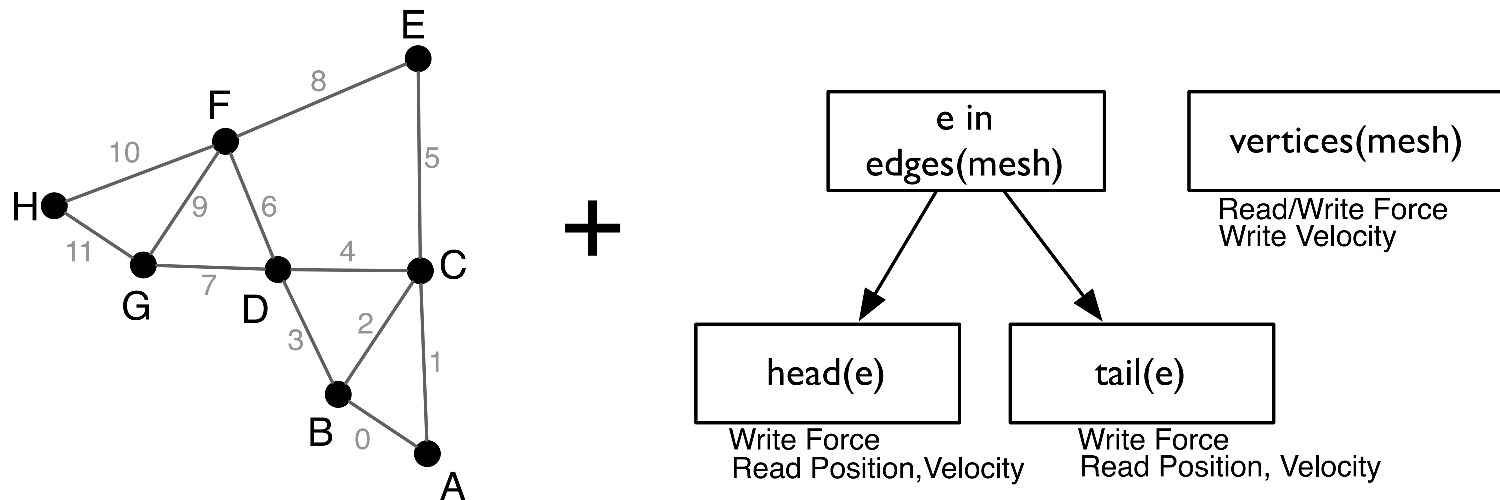

Ghost
Cells


Ghost
Cells

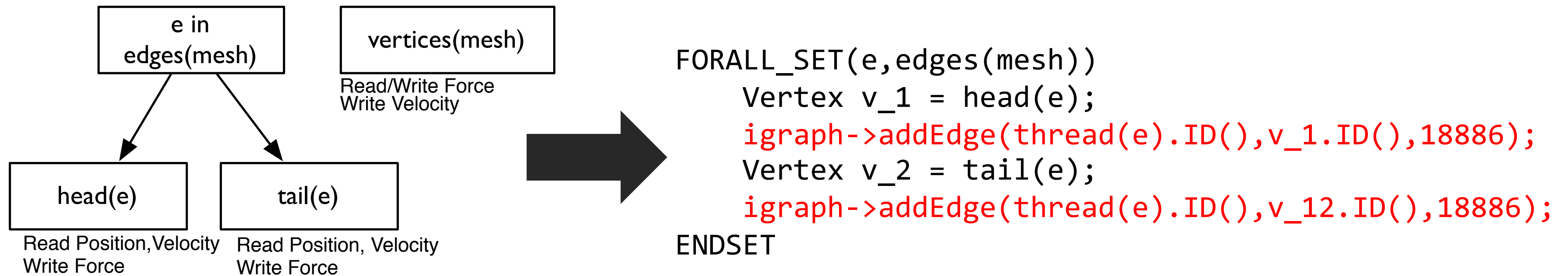
GPU: Schedule threads with coloring

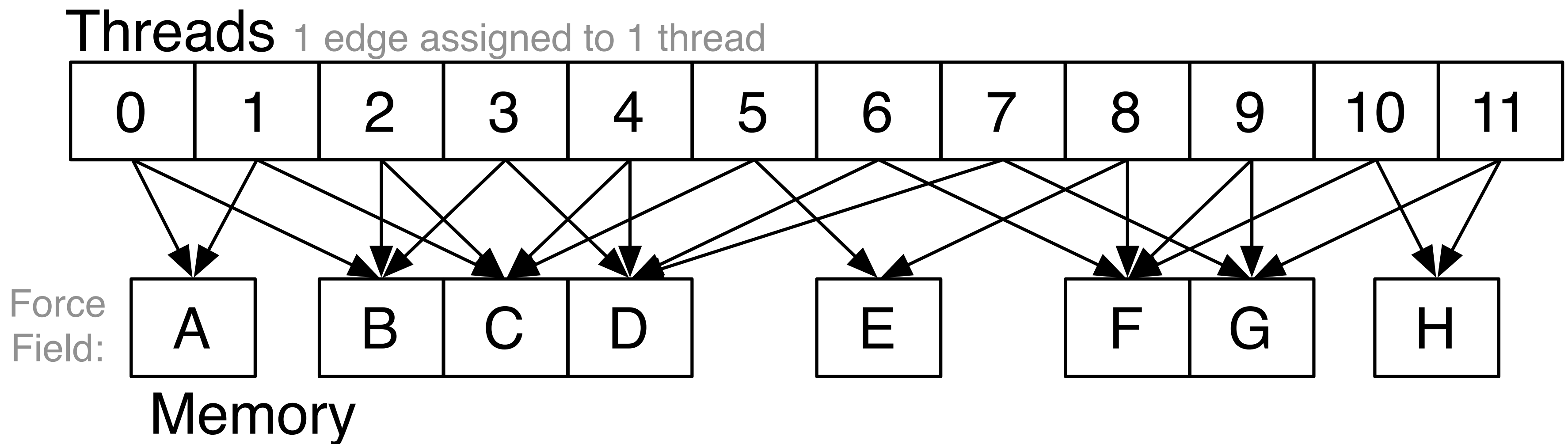
- Shared Memory
- Field updates need to be atomic
- MPI approach doesn't work – volume vs surface area

Build a graph of interfering writes:

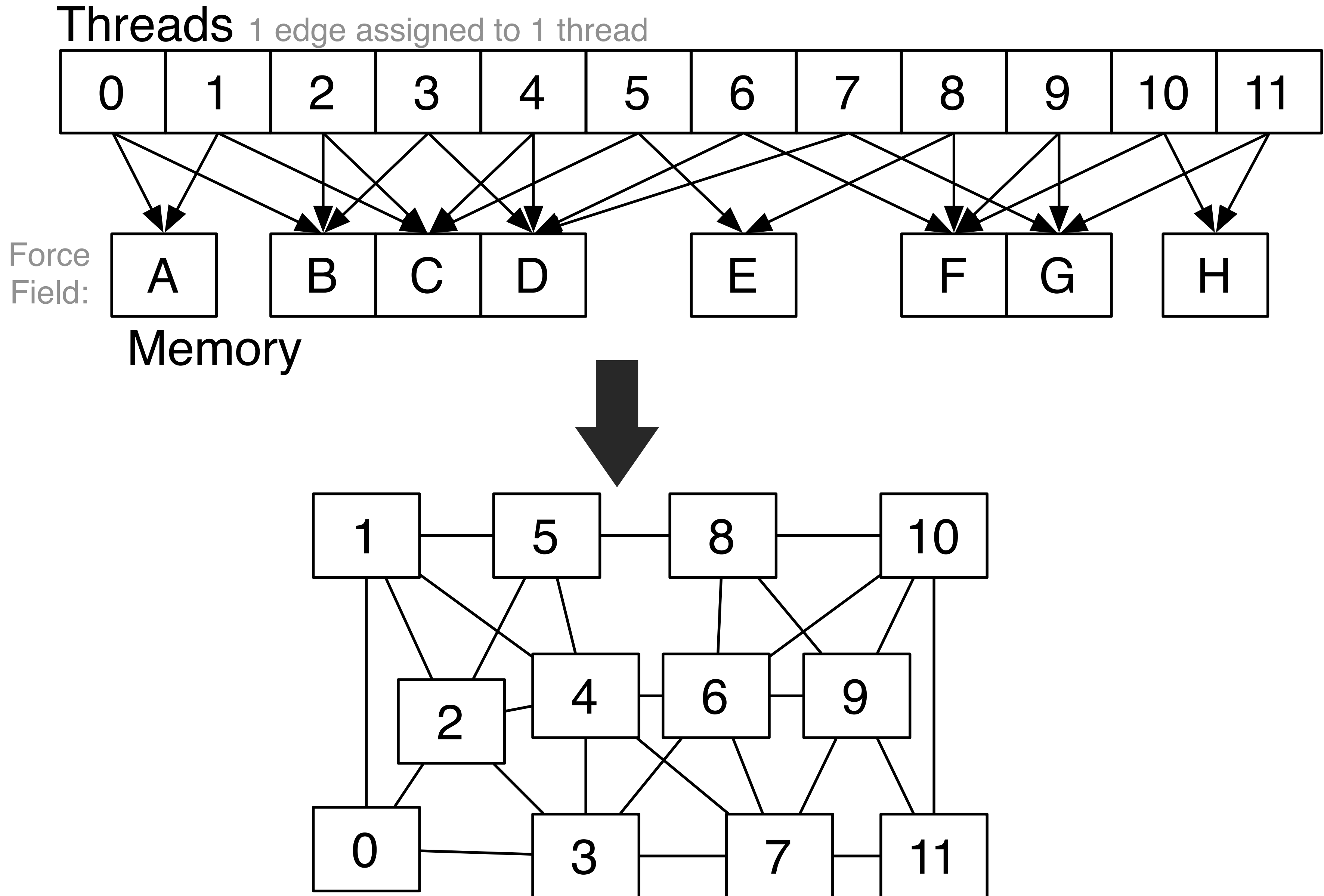


GPU: Schedule threads with coloring

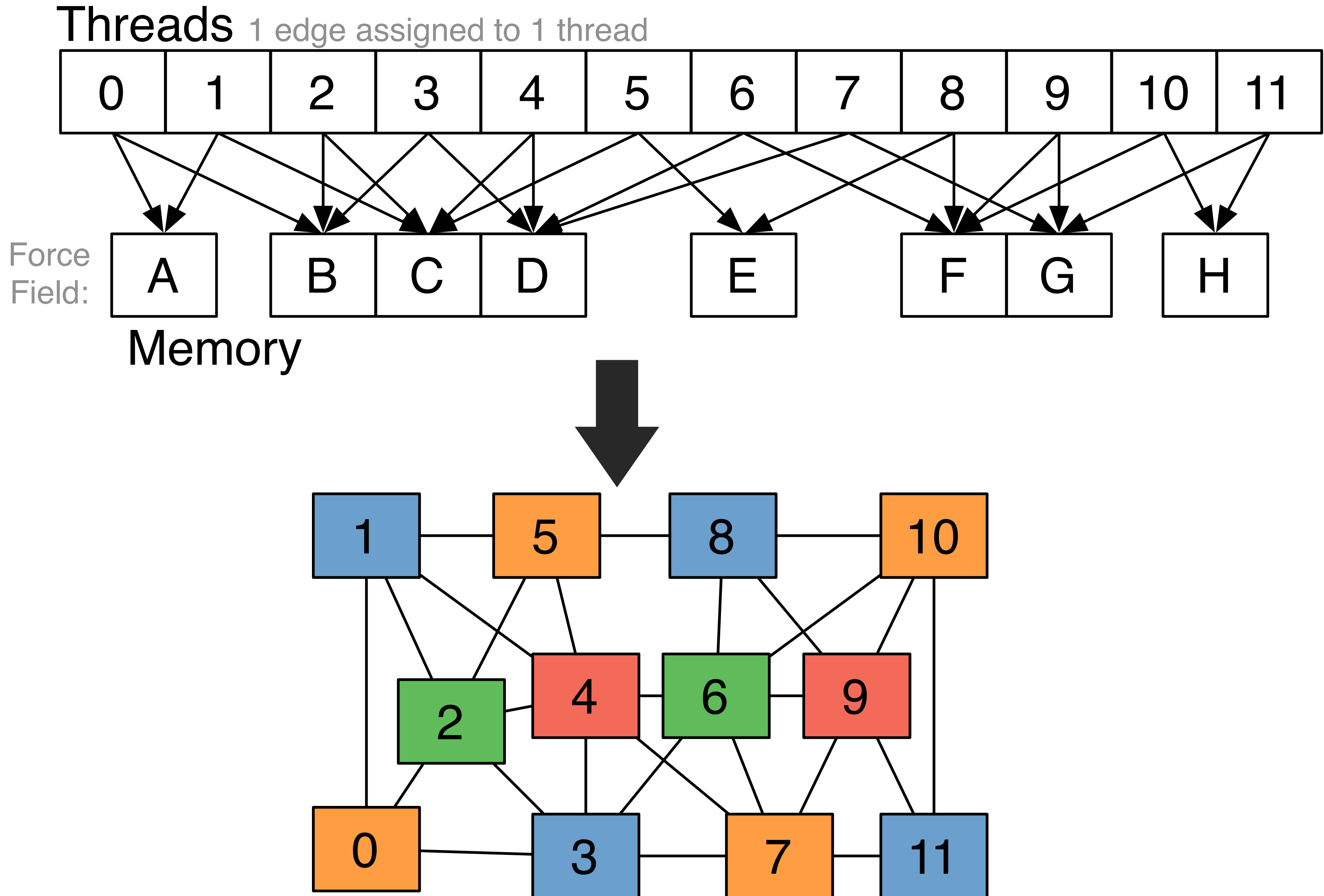




GPU: Schedule threads with coloring



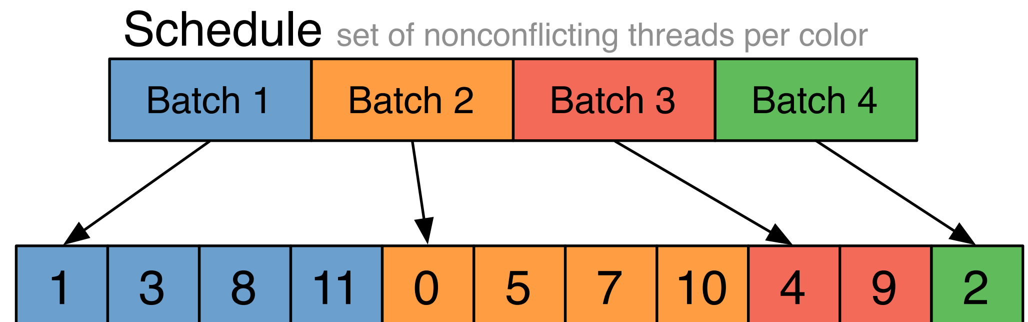
GPU: Schedule threads with coloring



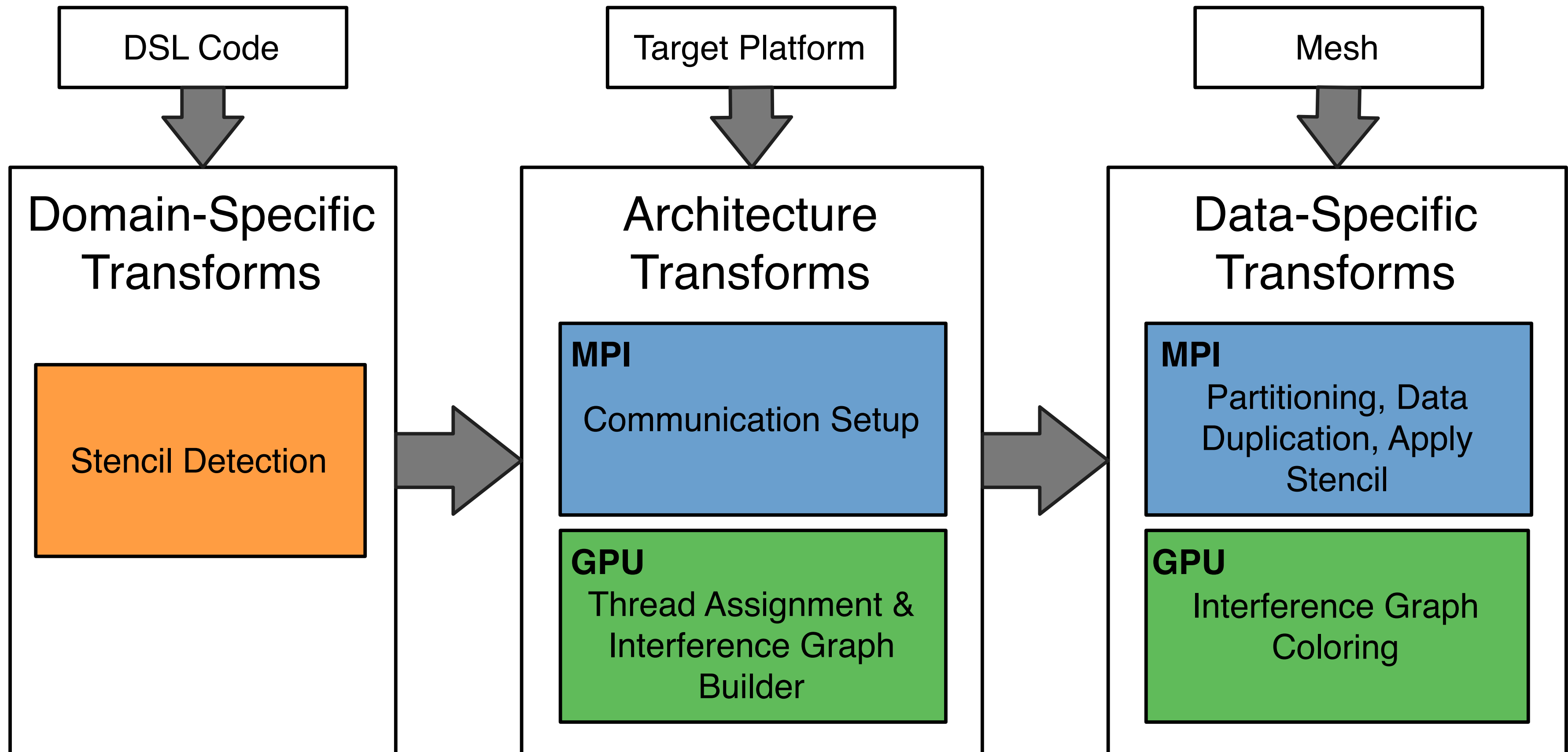
GPU: Schedule threads with coloring

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

```
__global__ for_01(ColorBatch batch, Force,
                  Position, Velocity, Maxforce) {
    val L = Position(v1) - Position(v2)
    ...
    Force(v1) += springForce
    Force(v2) -= springForce
    maxforce(e) = max(maxforce, 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);
}
```



One problem, 2 different approaches



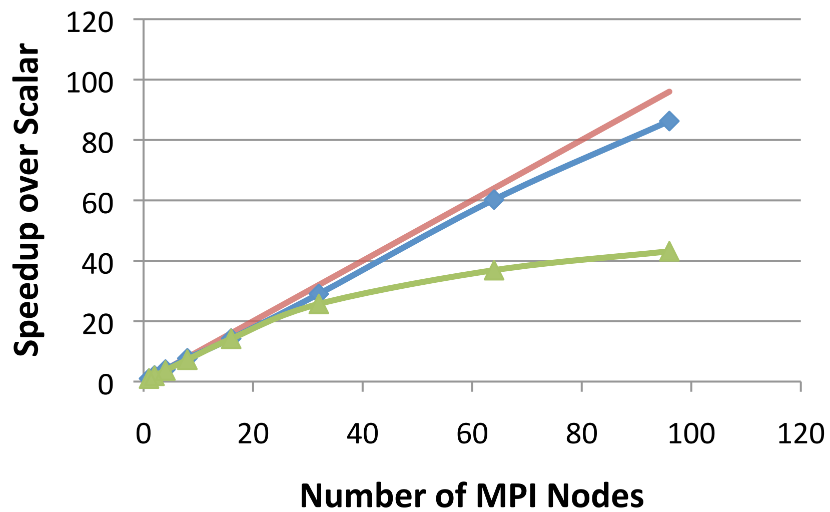
Evaluation

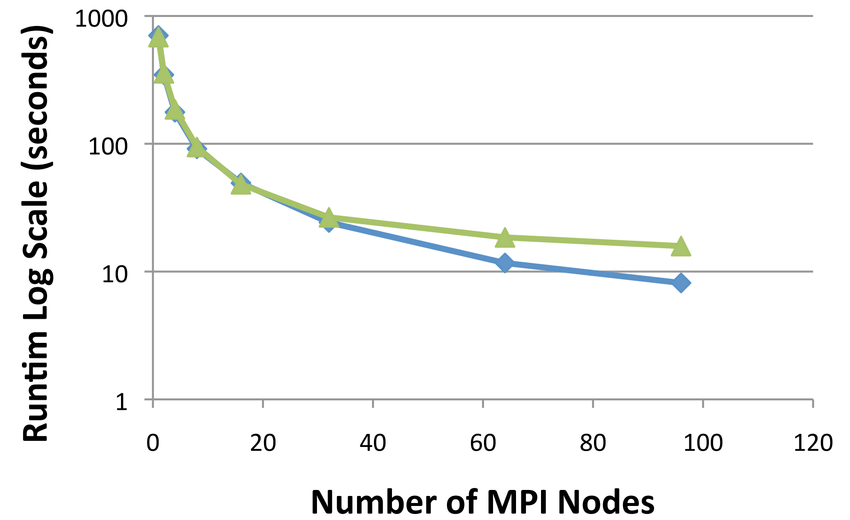
- Ported early version of Joe (A RANS Solver) to Liszt
- Time Integration Scheme: Forward Euler
- Mesh size: 750k cells

- MPI results:
  - Certainty Cluster – 4-socket 6-core 2.66Ghz Xeon CPU per node (24 cores), 36GB RAM per node.
- GPU Results:
  - Tesla C2050 (Fermi-based) GPU – 16 SMPs, 3GB GDDR5 RAM

MPI Performance

12 nodes, 8 cores per node

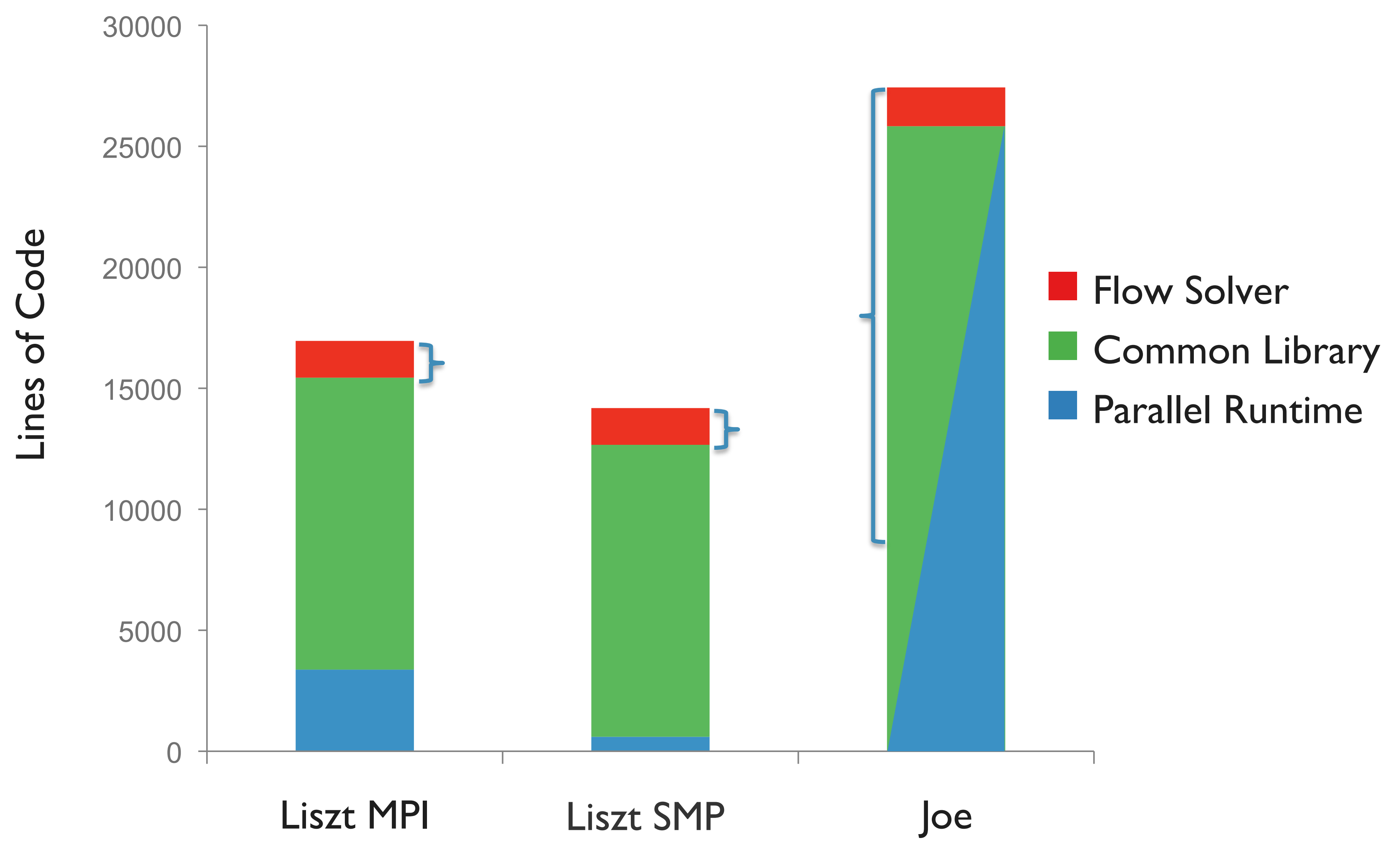




GPU Performance (Preliminary)

- Scaling mesh size from 50k (unit-sized) cells to 750k on a Tesla C2050.
- Comparison is against single threaded runtime on host CPU
- Double-precision: 18x
- Performance degraded by poor locality introduced by coloring
- Plan to fix with a 2-level blocking approach

Programmer Productivity



Four Advantages

Productivity

- Separate computational science expertise from computer science expertise

Portability

- Run on wide range of platforms

Performance

- Super-optimize using a combination of domain knowledge and platform knowledge

Innovation

- Allows vendors to change architecture and programming models in revolutionary ways

Four Advantages

Productivity

- Separate computational science expertise from computer science expertise

Portability

- Run o

Performance

- Super-optimize using a combination of domain knowledge and platform knowledge

Innovation

- Allows vendors to change architecture and programming models in revolutionary ways

`liszt.stanford.edu`

Criticisms of Domain-Specific Languages

Separate system with unique syntax

- Do I have to learn another language?

Usually a poor language designed by amateurs

- Why isn't it more general?

Difficult to integrate with other libraries/languages

- How do DSLs and non-DSLs interoperate?

No development environment

- Where is the debugger? Performance analysis tools?

Expensive to build complete compiler