Lecture 24:

Lizst Language Notes

Visual Computing Systems
CMU 15-769, Fall 2016

Slide acknowledgments:
Pat Hanrahan, Zach Devito (Stanford University)

What a Liszt program does

A Liszt program is run on a mesh

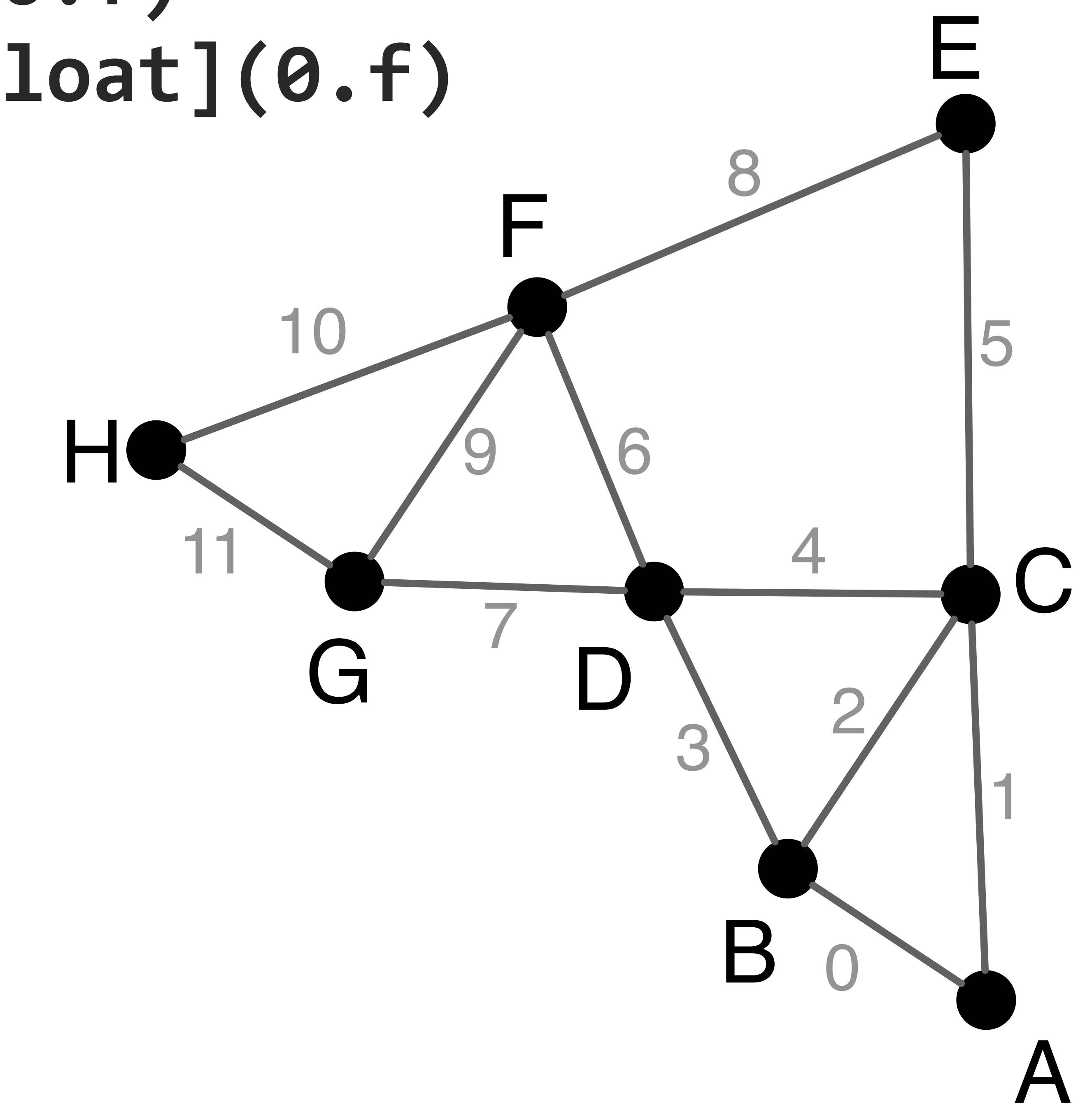
A Liszt program defines, and compute the value of, fields defined on the mesh

Position is a field defined at each mesh vertex.
The field's value is represented by a 3-vector.

```
val Position = FieldWithConst[Vertex,Float3](0.f, 0.f, 0.f)
val Temperature = FieldWithConst[Vertex,Float](0.f)
val Flux = FieldWithConst[Vertex,Float](0.f)
val JacobiStep = FieldWithConst[Vertex,Float](0.f)
```

Color key:

Fields
Mesh entity



Notes:
Fields are a higher-kinded type
(special function that maps a type to a new type)

Liszt program: heat conduction on mesh

Program computes the value of fields defined on meshes

```
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
}
```

Set flux for all vertices to 0.f;

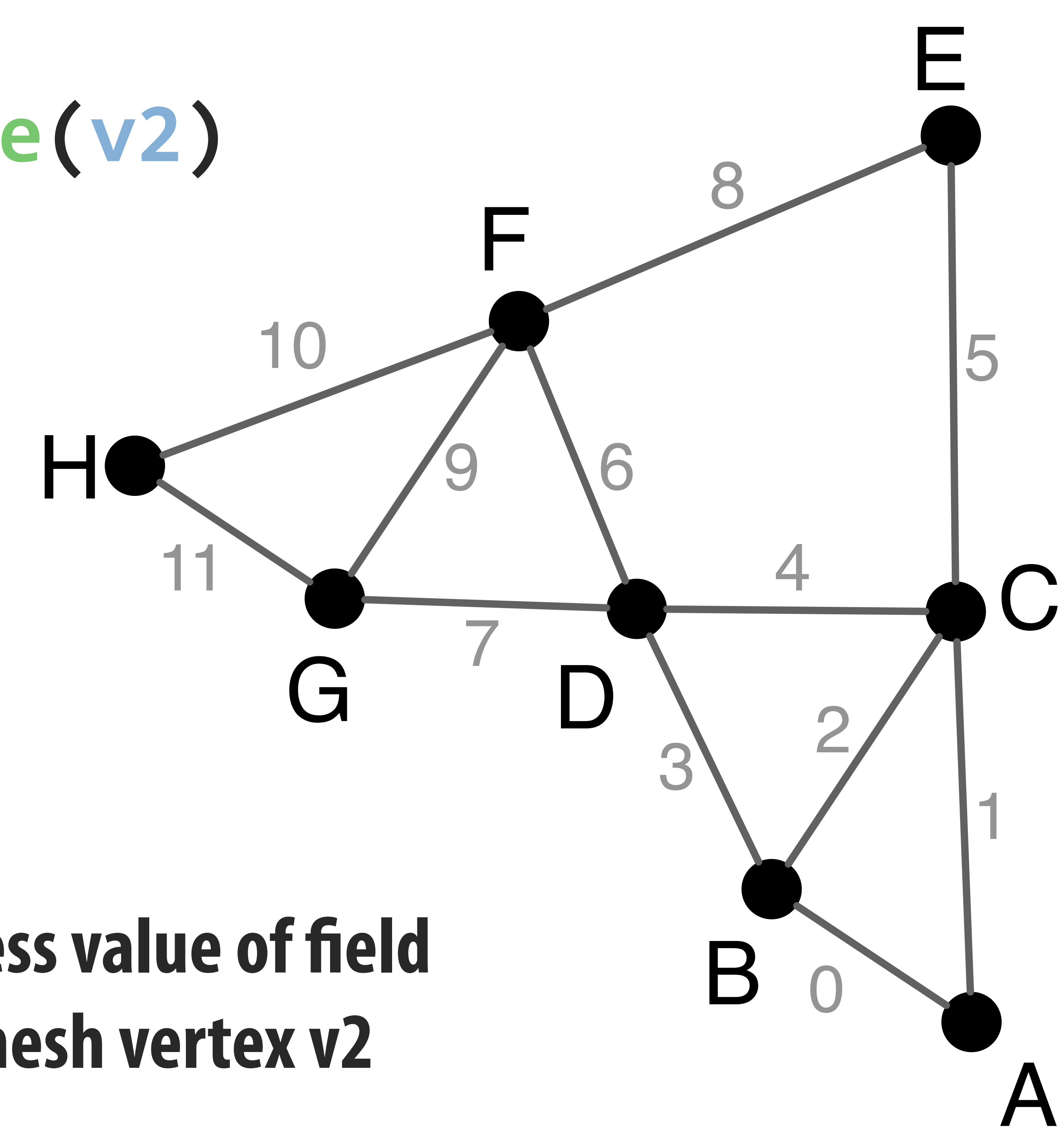
Independently, for each edge in the mesh

Given edge, loop body accesses/modifies field values at adjacent mesh vertices

Access value of field at mesh vertex v2

Color key:

- Fields
- Mesh
- Topology functions
- Iteration over set



Liszt's topological operators

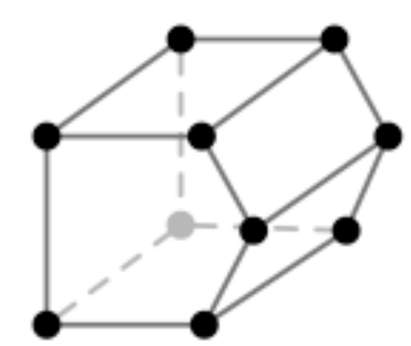
Used to access mesh elements relative to some input vertex, edge, face, etc.

Topological operators are the <u>only way</u> to access mesh data in a Liszt program

Notice how many operators return sets (e.g., "all edges of this face")

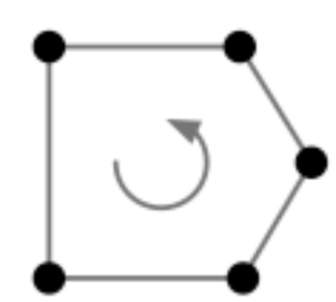
```
BoundarySet[1][ME <: MeshElement](name : String) : Set[ME]
vertices(e : Mesh) : Set[Vertex]
cells(e : Mesh) : Set[Cell]
edges(e : Mesh) : Set[Edge]
faces(e : Mesh) : Set[Face]
```

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



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

```
vertices(e : Edge) : Set[Vertex]
facesCCW[2](e : Edge) : Set[Face]
cells(e : Edge) : Set[Cell]
head(e : Edge) : Vertex
tail(e : Edge) : Vertex
flip[4](e : Edge) : Edge
towards[5](e : Edge, t : Vertex) : Edge
```



```
cells(e : Face) : Set[Cell]
edgesCCW[2](e : Face) : Set[Edge]
vertices(e : Face) : Set[Vertex]
inside[3](e : Face) : Cell
outside[3](e : Face) : Cell
flip[4](e : Face) : Face
towards[5](e : Face,t : Cell) : Face
```

Liszt programming

- A Liszt program describes operations on fields of an abstract mesh representation
- Application specifies type of mesh (regular, irregular) and its topology
- Mesh representation is chosen by Liszt (not by the programmer)
  - Based on mesh type, program behavior, and target machine

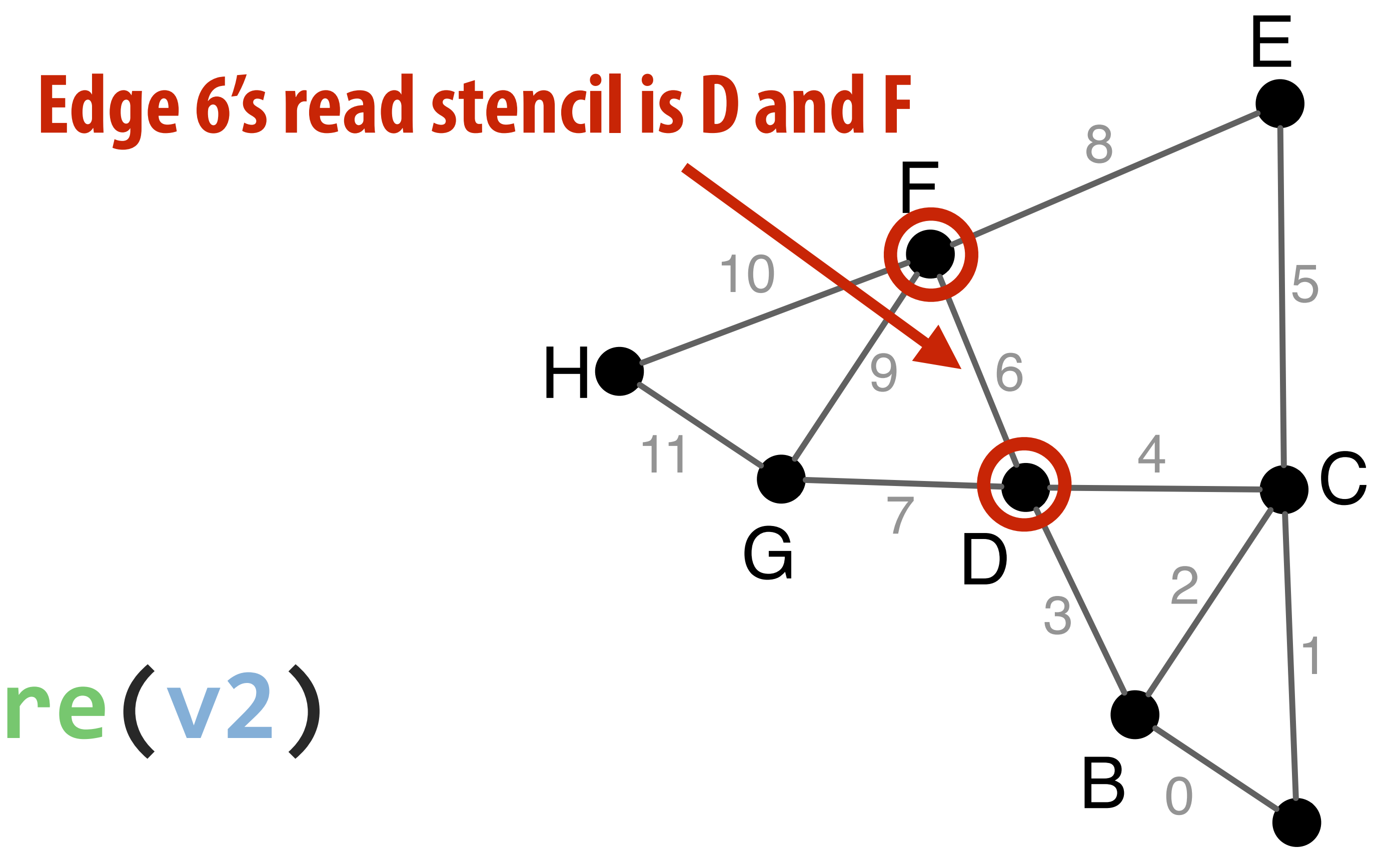
Well, that's interesting. I write a program, and the compiler decides what data structure it should use based on what operations my code performs.

Liszt is constrained to allow dependency analysis

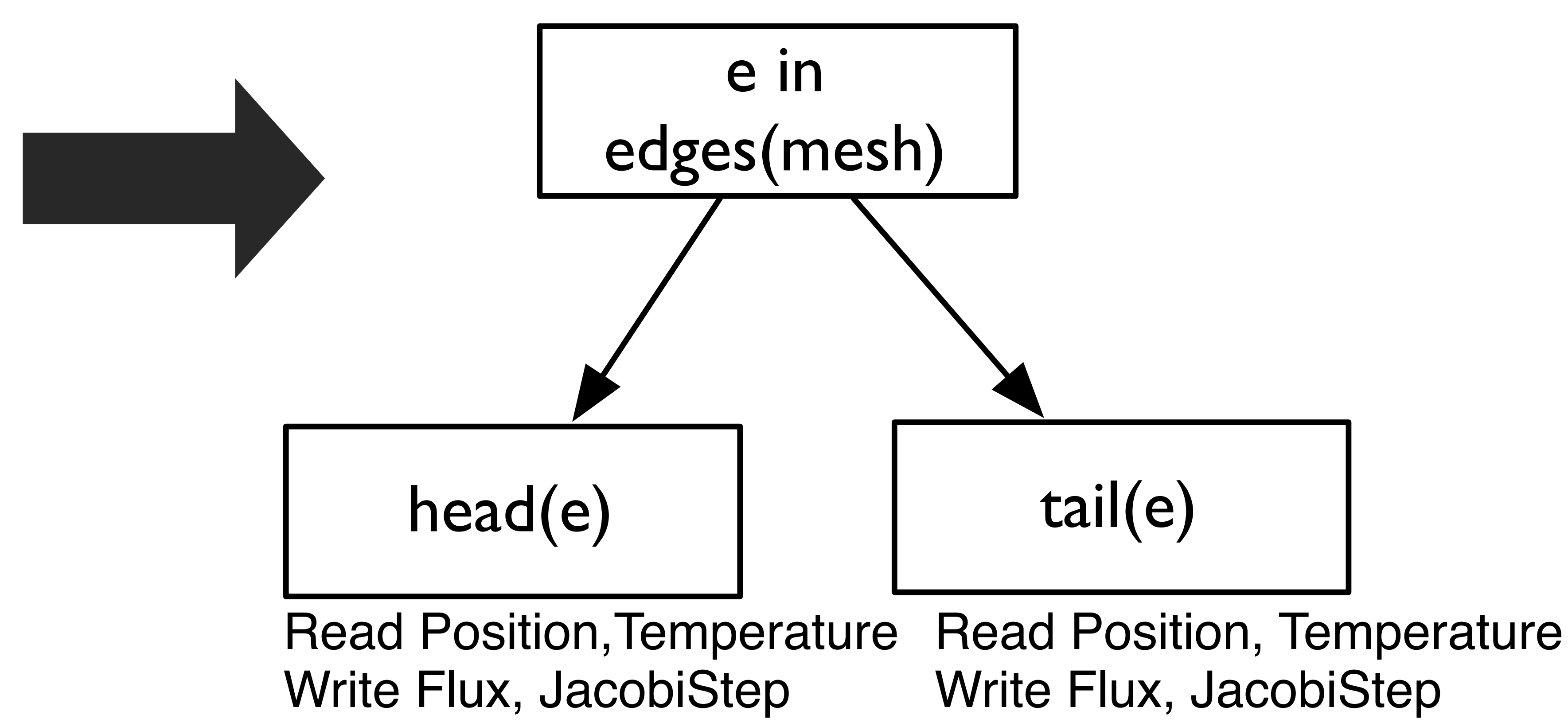
Lizst infers "stencils": "stencil" = mesh elements accessed in an iteration of loop
= dependencies for the iteration

Statically analyze code to find stencil of each top-level **for** loop

- Extract nested mesh element reads
- Extract field operations




```
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
}
...
```



Restrict language for dependency analysis

Language restrictions:

- Mesh elements are only accessed through built-in topological functions:

```
cells(mesh), …
```

- Single static assignment:

```
val v1 = head(e)
```

- Data in fields can only be accessed using mesh elements:

```
Pressure(v)
```

- No recursive functions

Restrictions allow compiler to automatically infer stencil for a loop iteration.

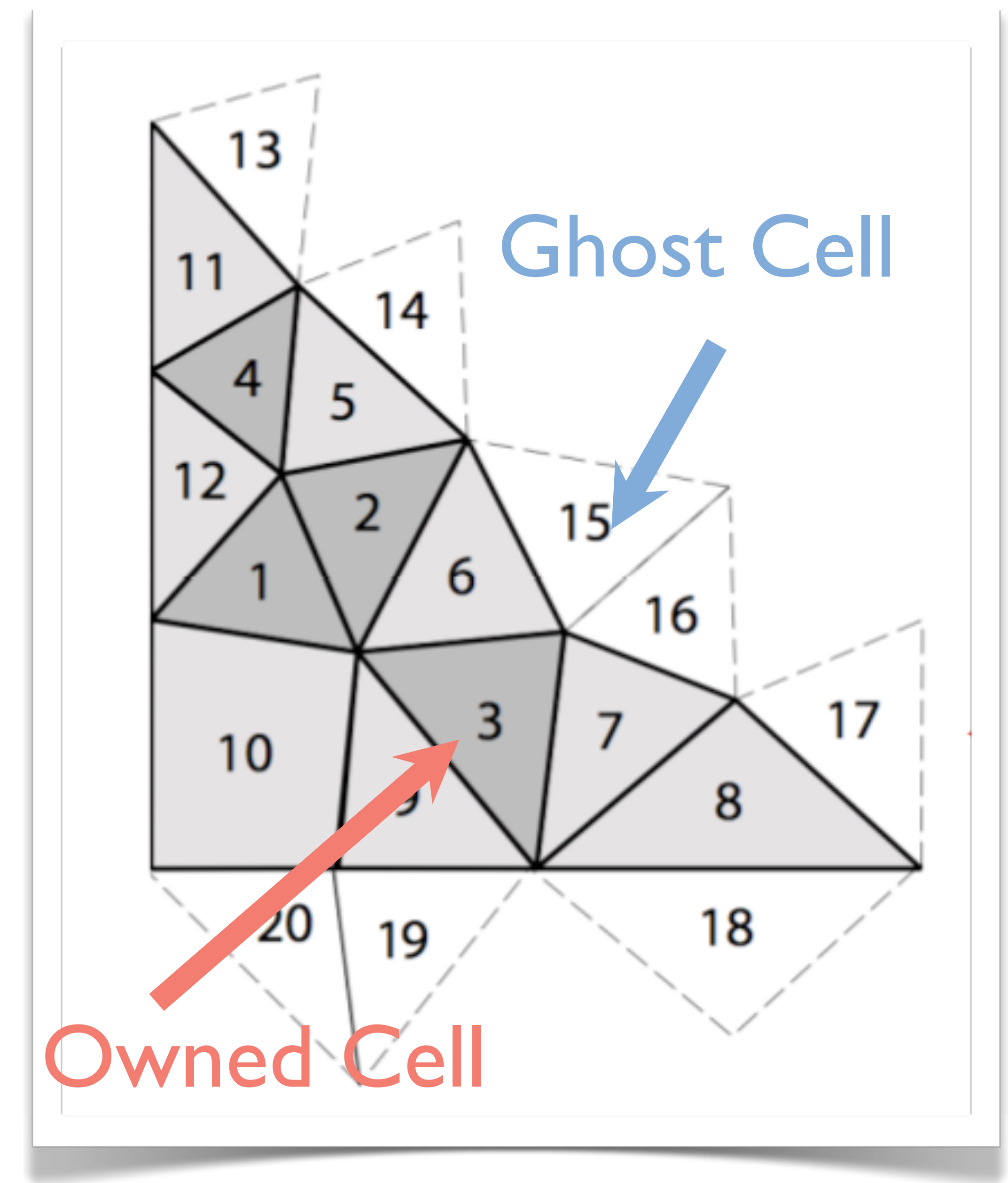
(Same idea as constraints that enable bounds analysis in Halide.)

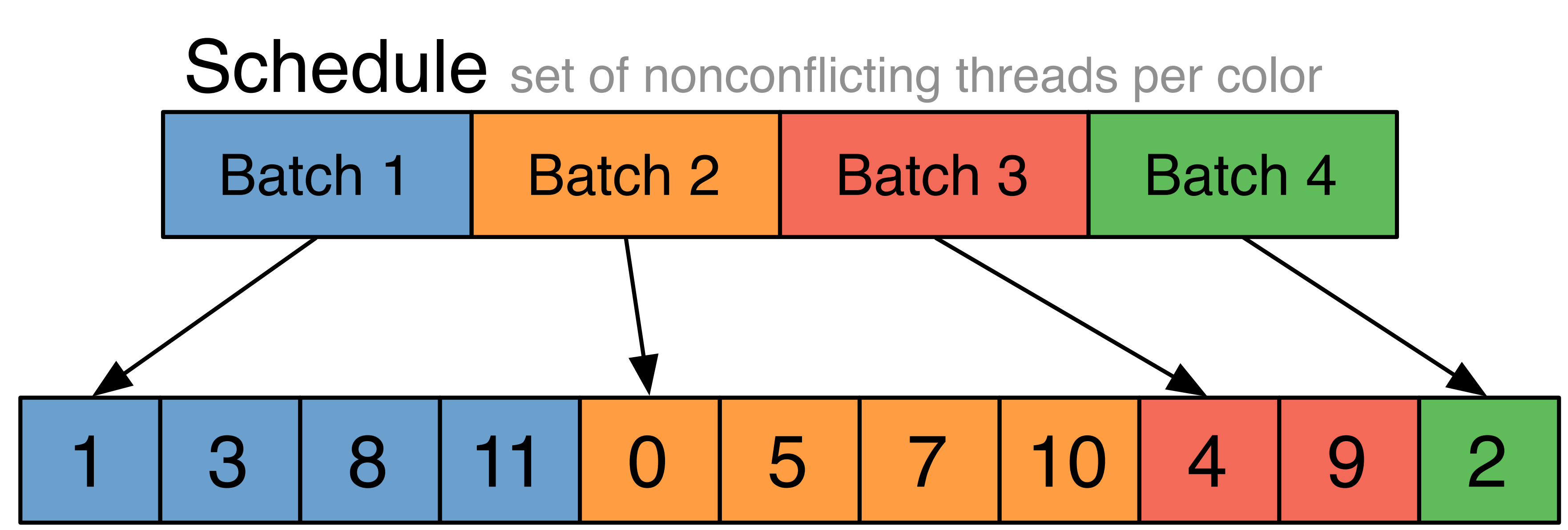
Portable parallelism: use dependencies to implement different parallel execution strategies

I'll discuss two strategies…

Strategy 1: mesh partitioning

Strategy 2: mesh coloring

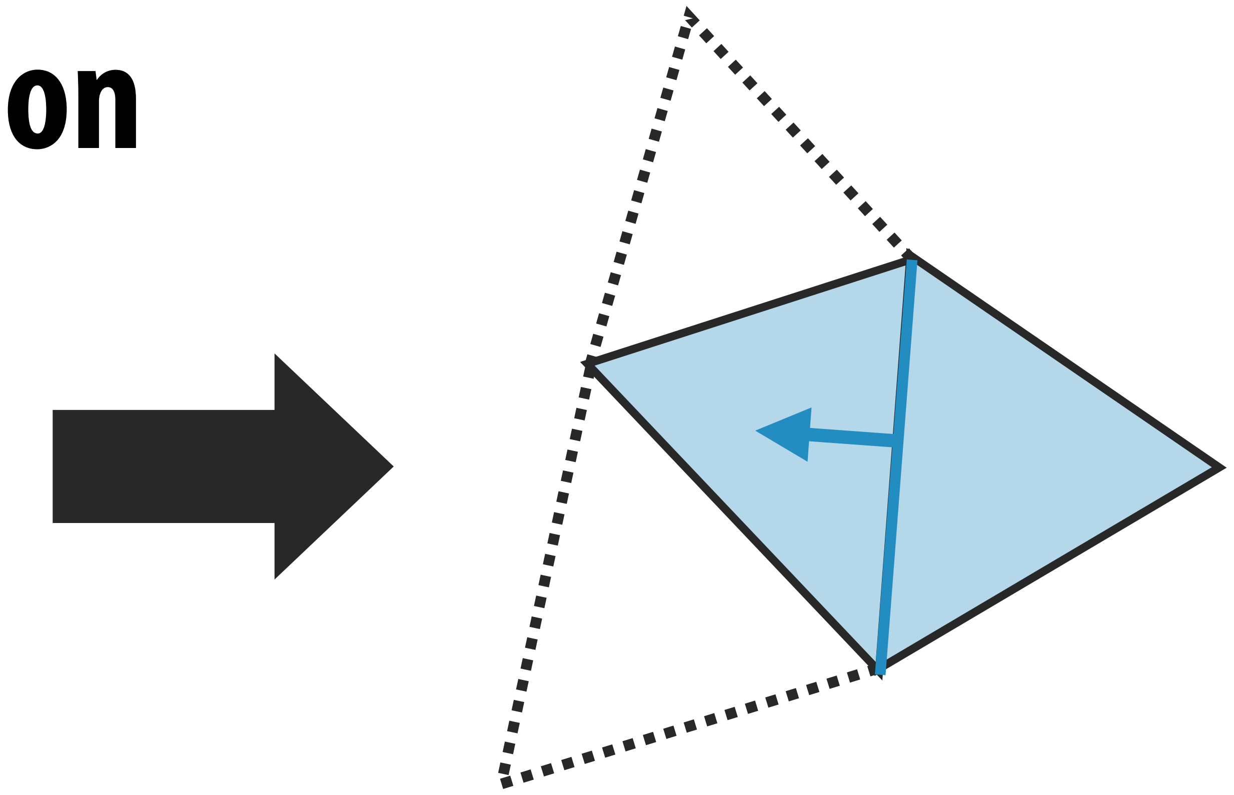


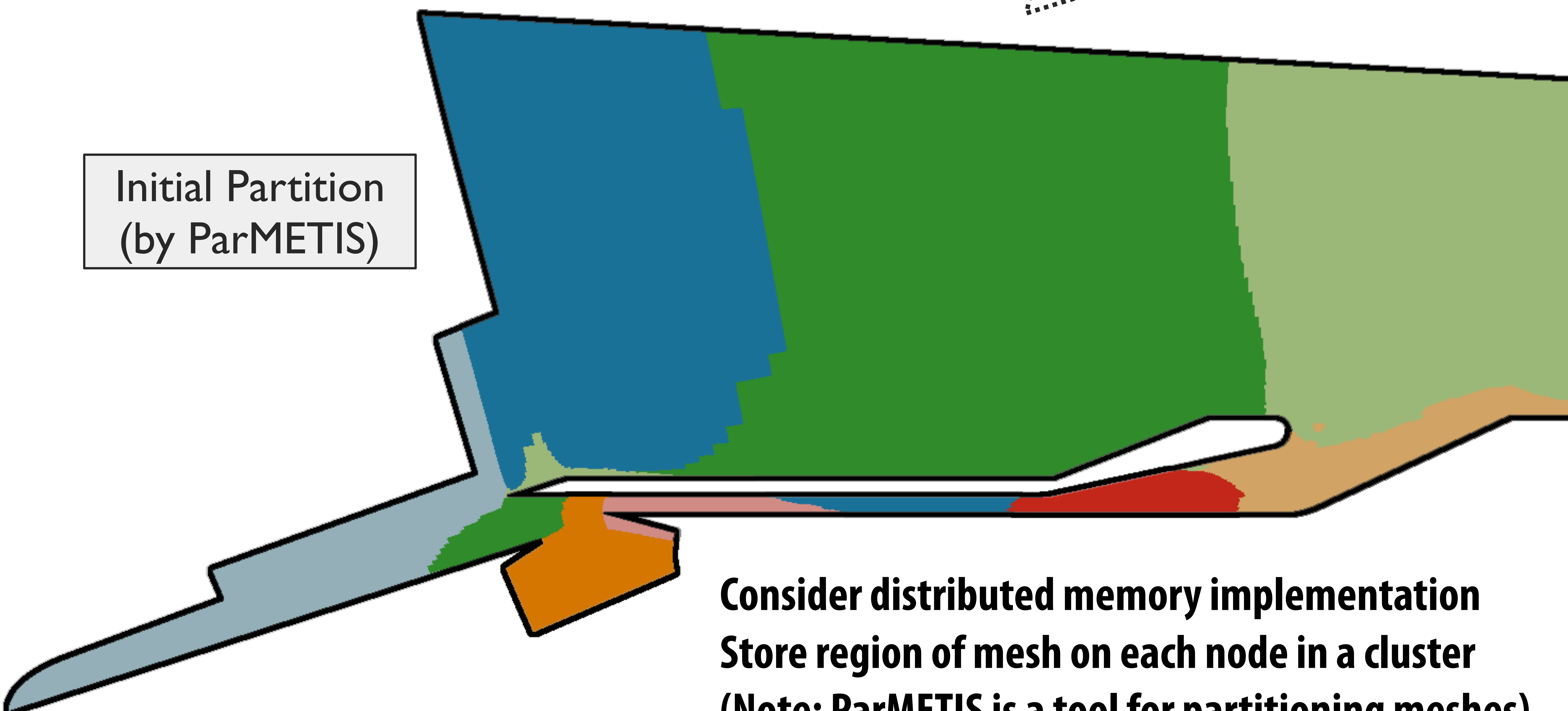


Distributed memory implementation of Liszt

Mesh + Stencil → Graph → Partition

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

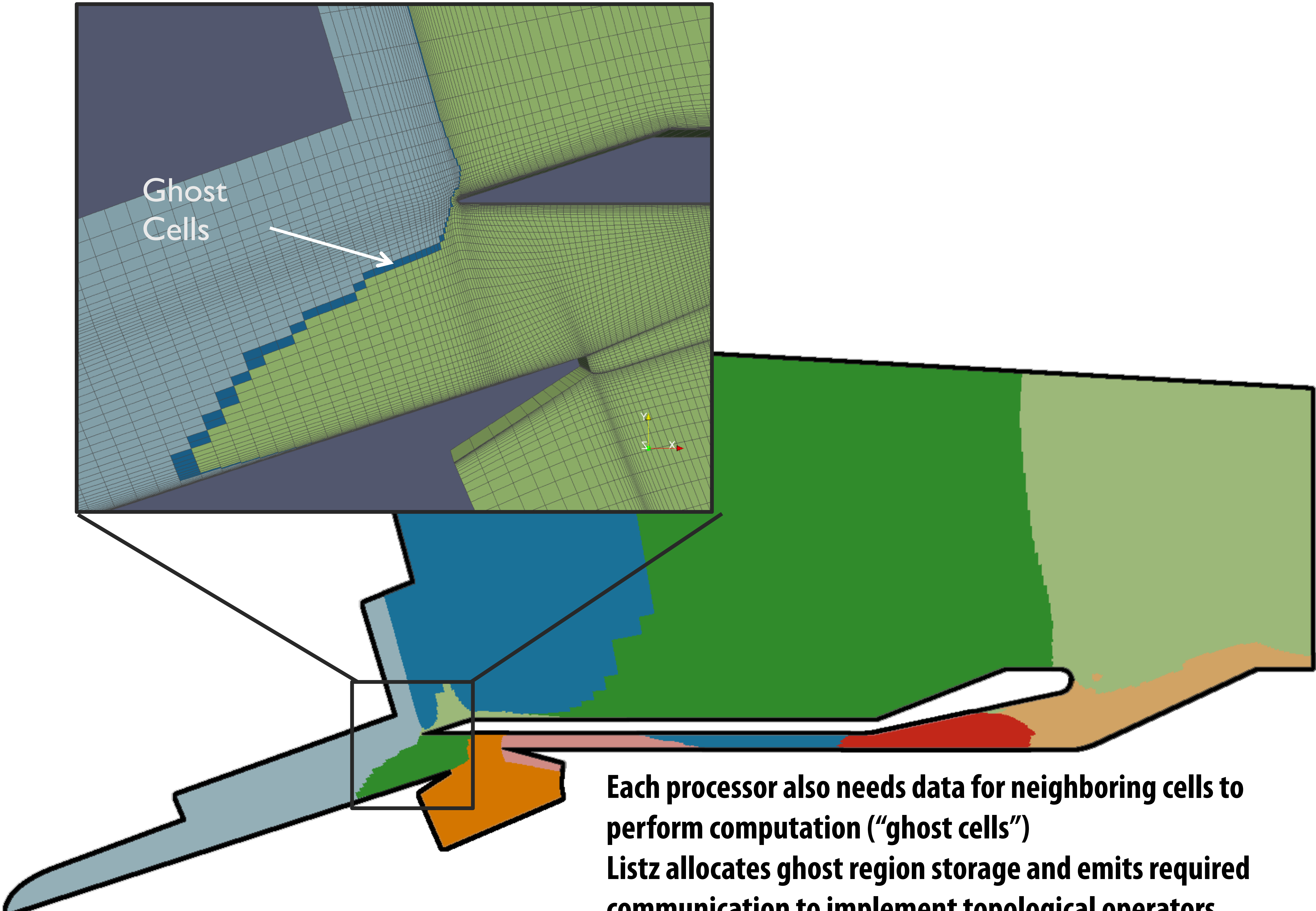




Consider distributed memory implementation

Store region of mesh on each node in a cluster

(Note: ParMETIS is a tool for partitioning meshes)



Each processor also needs data for neighboring cells to perform computation ("ghost cells")
Listz allocates ghost region storage and emits required communication to implement topological operators.

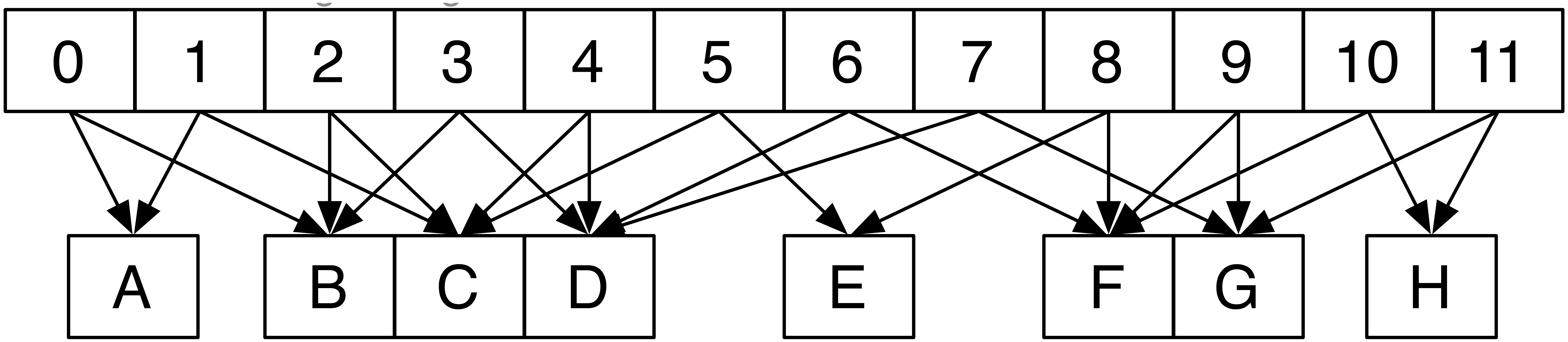
Imagine compiling a Lizst program to a GPU
(single address space, many tiny threads)

GPU implementation: parallel reductions

In previous example, one region of mesh assigned per processor (or node in MPI cluster)

On GPU, natural parallelization is one edge per CUDA thread

Threads (each edge assigned to 1 CUDA thread)



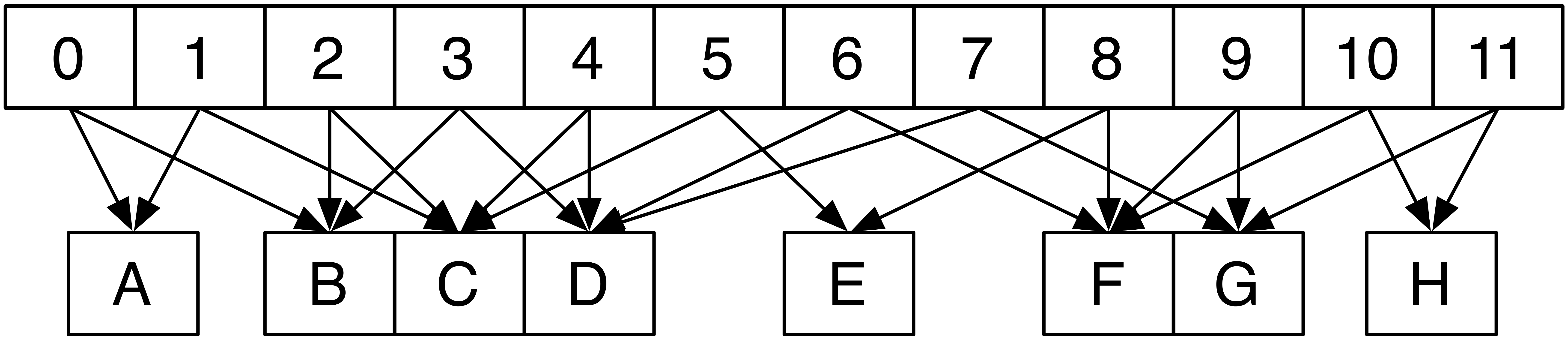
Flux field values (per vertex)

```
for (e <- edges(mesh)) {
  ...
  Flux(v1) += dT*step
  Flux(v2) -= dT*step
  ...
}
```

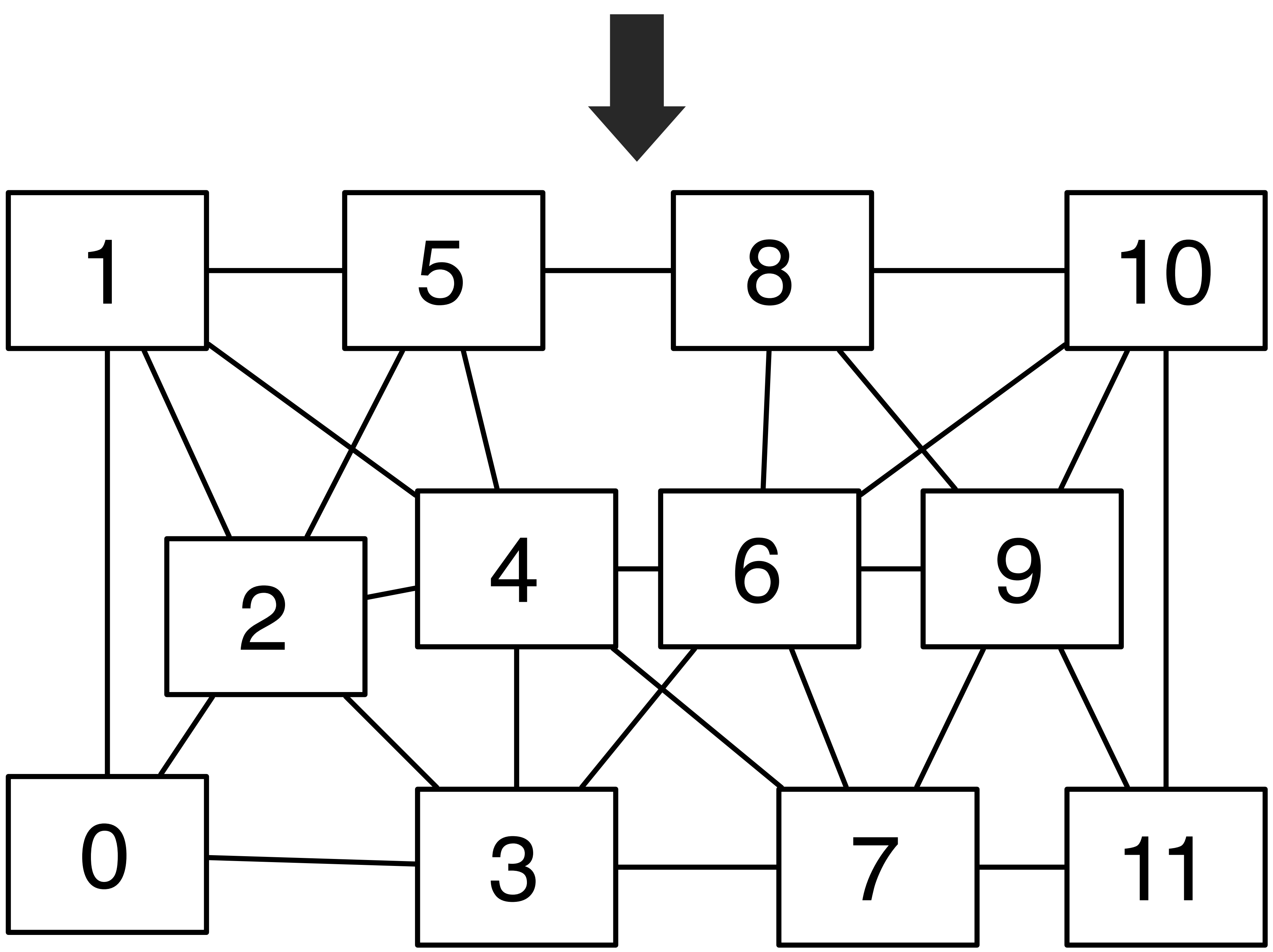
Different edges share a vertex: requires atomic update of per-vertex field data

GPU implementation: conflict graph

Threads (each edge assigned to 1 CUDA thread)



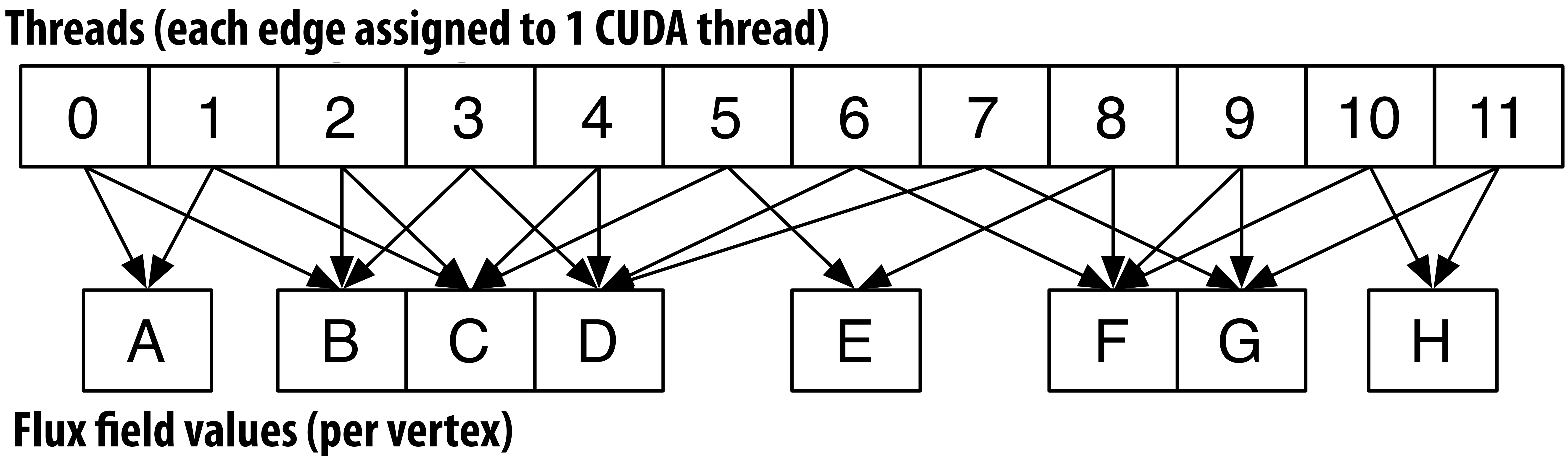
Flux field values (per vertex)

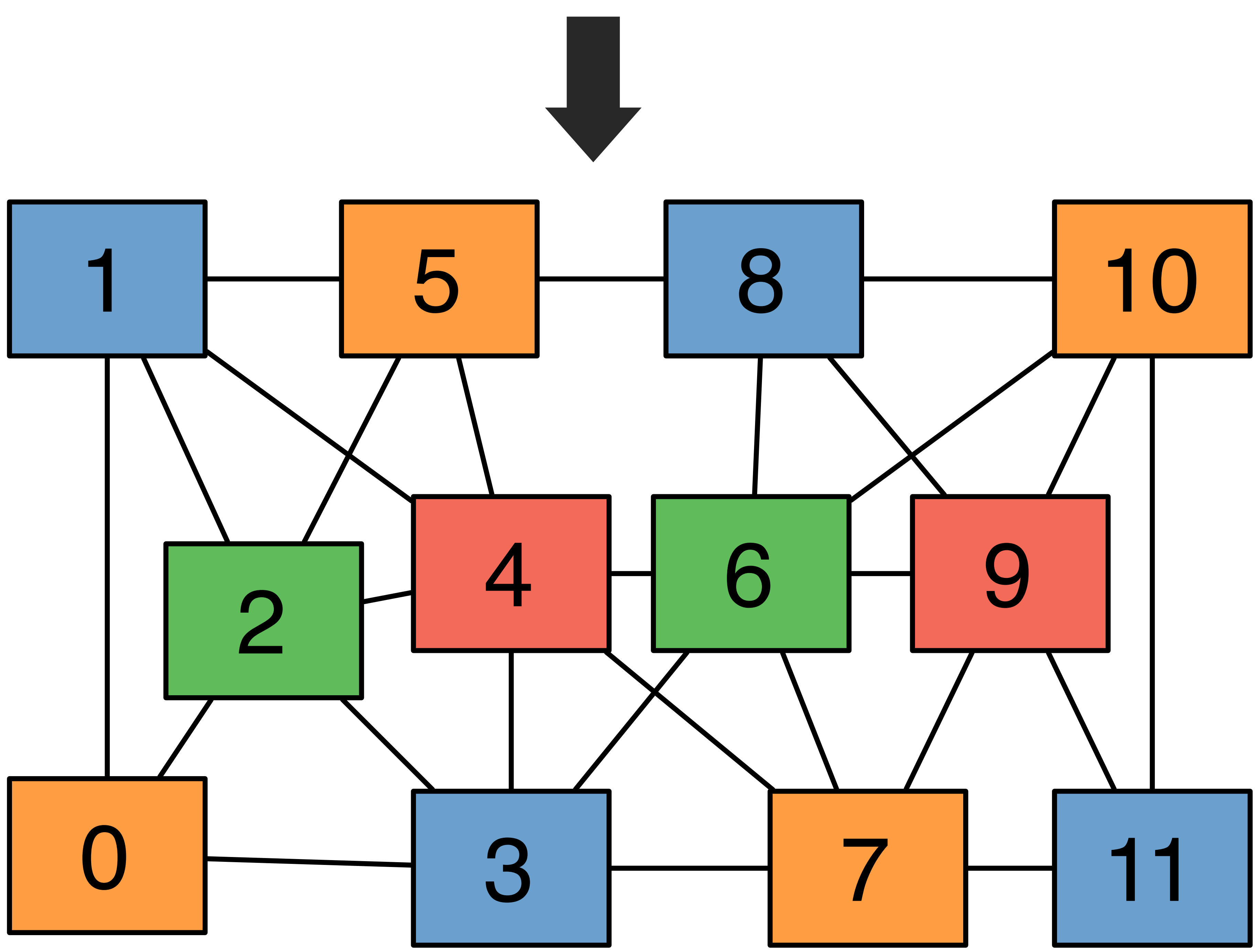


Identify mesh edges with colliding writes (lines in graph indicate presence of collision)

Can simply run program once to get this information.
(results valid for subsequent executions provided mesh does not change)

GPU implementation: conflict graph



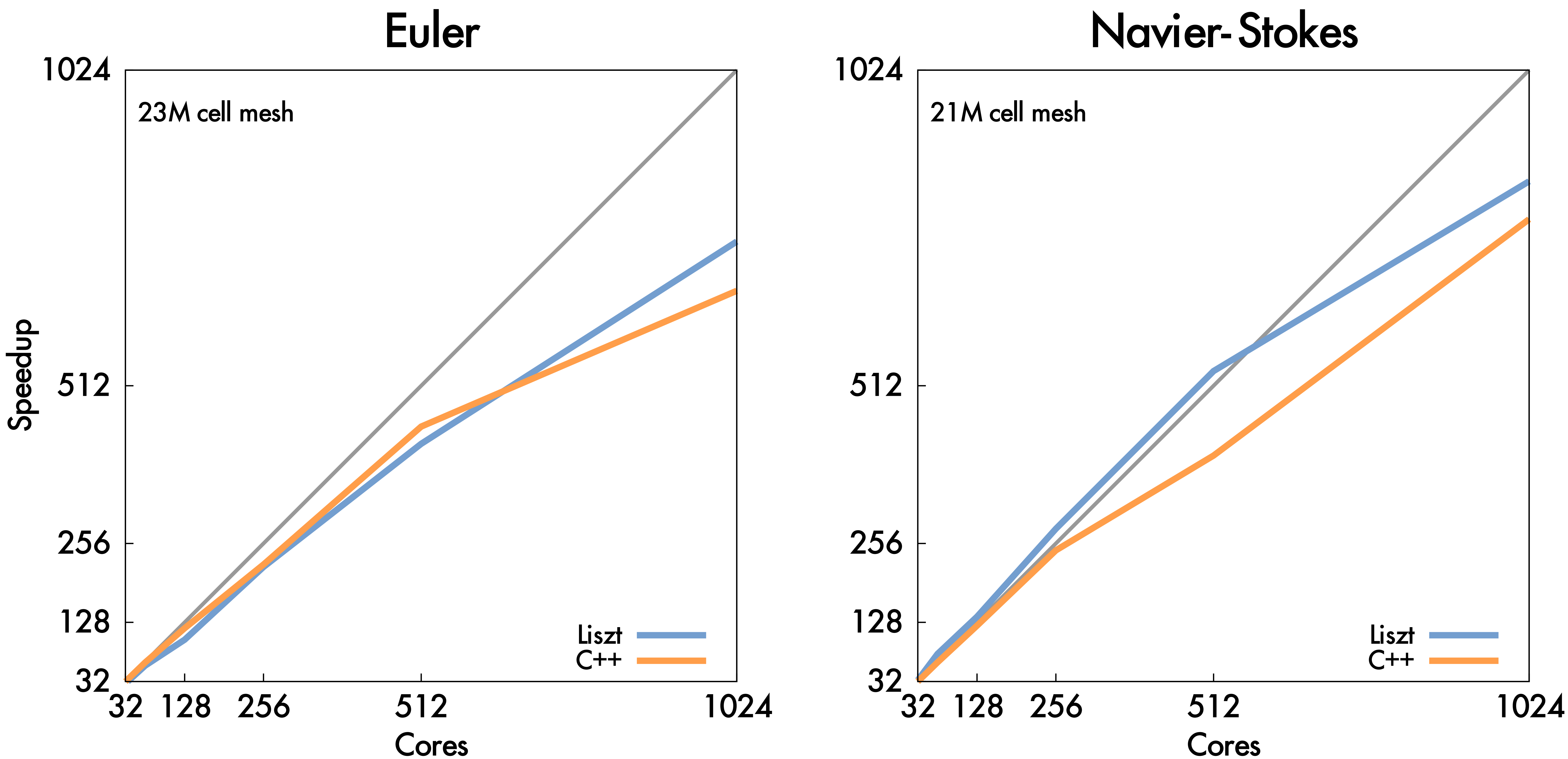


"Color" nodes in graph such that no connected nodes have the same color

Can execute on GPU in parallel, without atomic operations, by running all nodes with the same color in a single CUDA launch.

Cluster performance of Lizst program

256 nodes, 8 cores per node (message-passing implemented using MPI)



Important: performance portability!

Same Liszt program also runs with high efficiency on GPU (results not shown here).

But uses a <u>different algorithm</u> when compiled to GPU! (graph coloring)

Liszt summary

- **Productivity:**
  - Abstract representation of mesh: vertices, edges, faces, fields (concepts that a scientist thinks about already!)
  - Intuitive topological operators
- **Portability**
  - Same code runs on large cluster of CPUs (MPI) and GPUs (and combinations thereof!)
- **High-performance**
  - Language is constrained to allow compiler to track dependencies
  - Used for locality-aware partitioning in distributed memory implementation
  - Used for graph coloring in GPU implementation
  - Compiler knows how to chooses different parallelization strategies for different platforms
  - Underlying mesh representation can be customized by system based on usage and platform (e.g, don't store edge pointers if code doesn't need it, choose struct of arrays vs. array of structs for per-vertex fields)

Class discussion on Ebb
(Bernstein et al. SIGGRAPH 16)