ICSE
THE INSTITUTE FOR CLEAN AND SECURE ENERGY
▪155 South 1452 East Room 380 ▪Salt Lake City, Utah 84112 ▪1-801-585-1233

Graph-Based Parallel Task Scheduling and Algorithm Generation for Multiphysics PDE Software

James C. Sutherland
ASSISTANT PROFESSOR - CHEMICAL ENGINEERING

Devin Robison
M.S. STUDENT

This research was sponsored by the National Nuclear Security Administration under the Accelerating Development of Retrofitable CO2 Capture Technologies through Predictivity program through DOE Cooperative Agreement DE-NA0000740

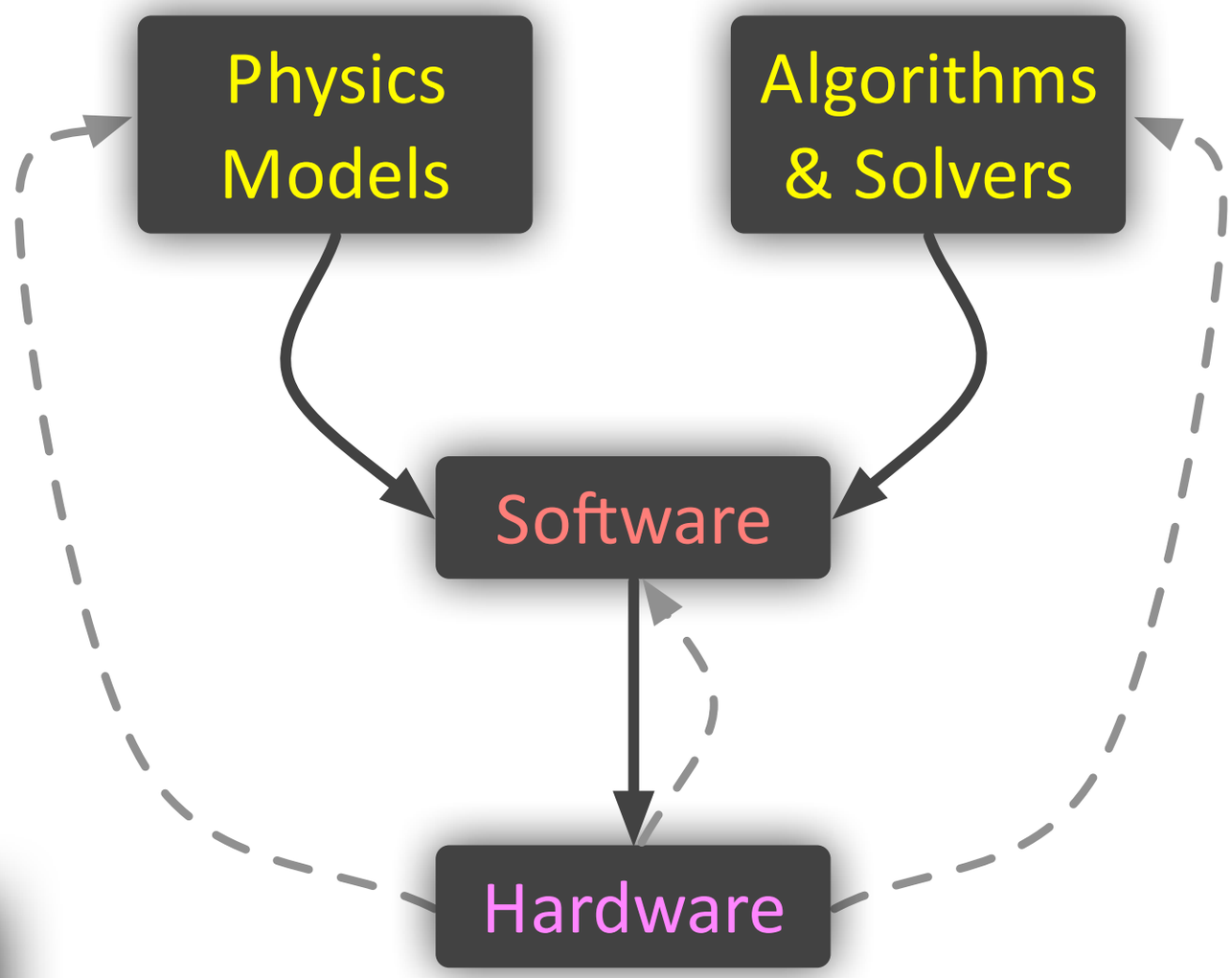
This work was funded in part by NSF PetaApps award 0904631
National Science Foundation WHERE DISCOVERIES BEGIN

Motivation

Physics is complex & changing

Software must handle complexity & changing nature of:

- physics models,
- algorithms,
- hardware

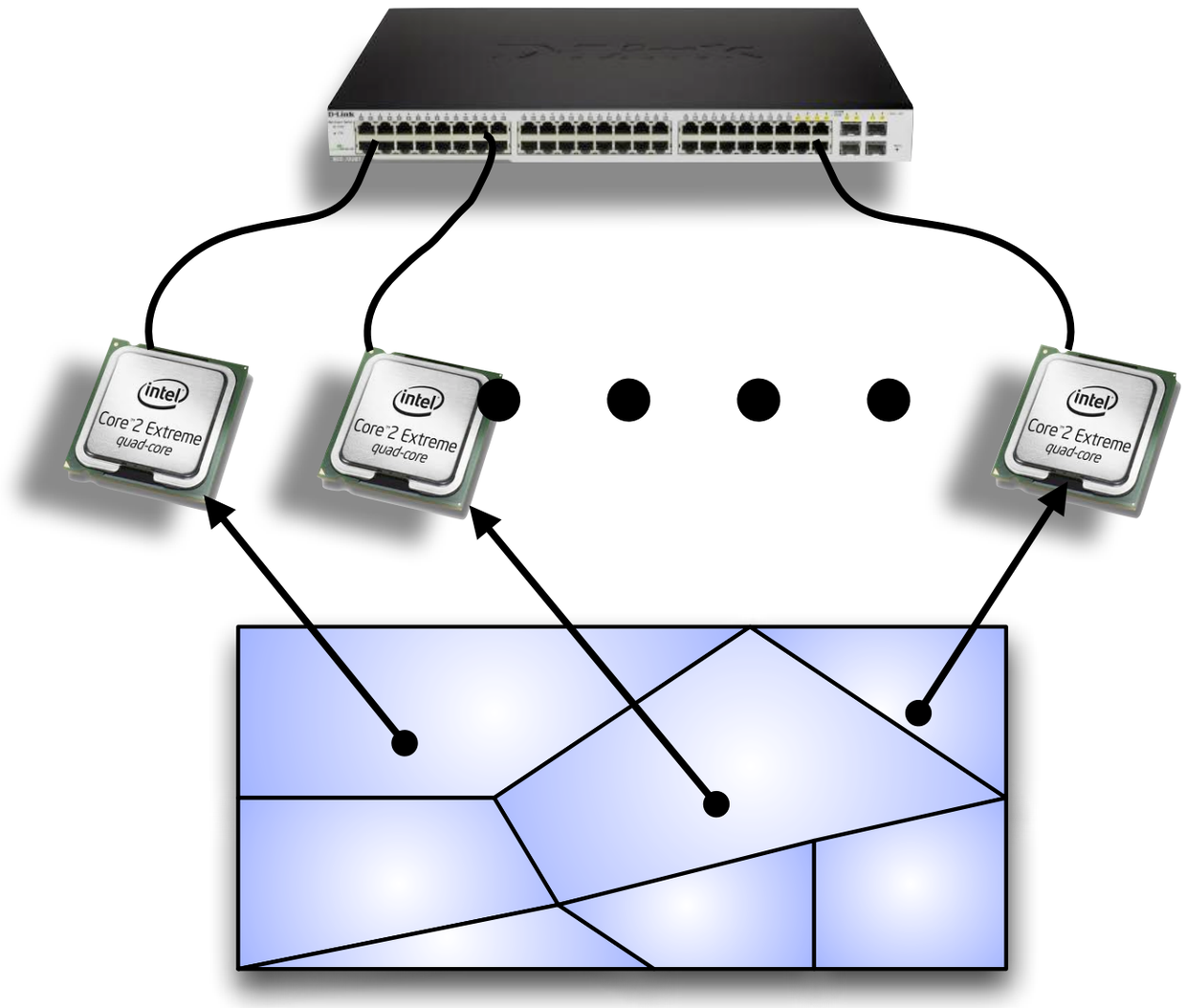


Hardware is complex & changing

- muticore/manycore
- hybrid CPU/GPU or other accelerators

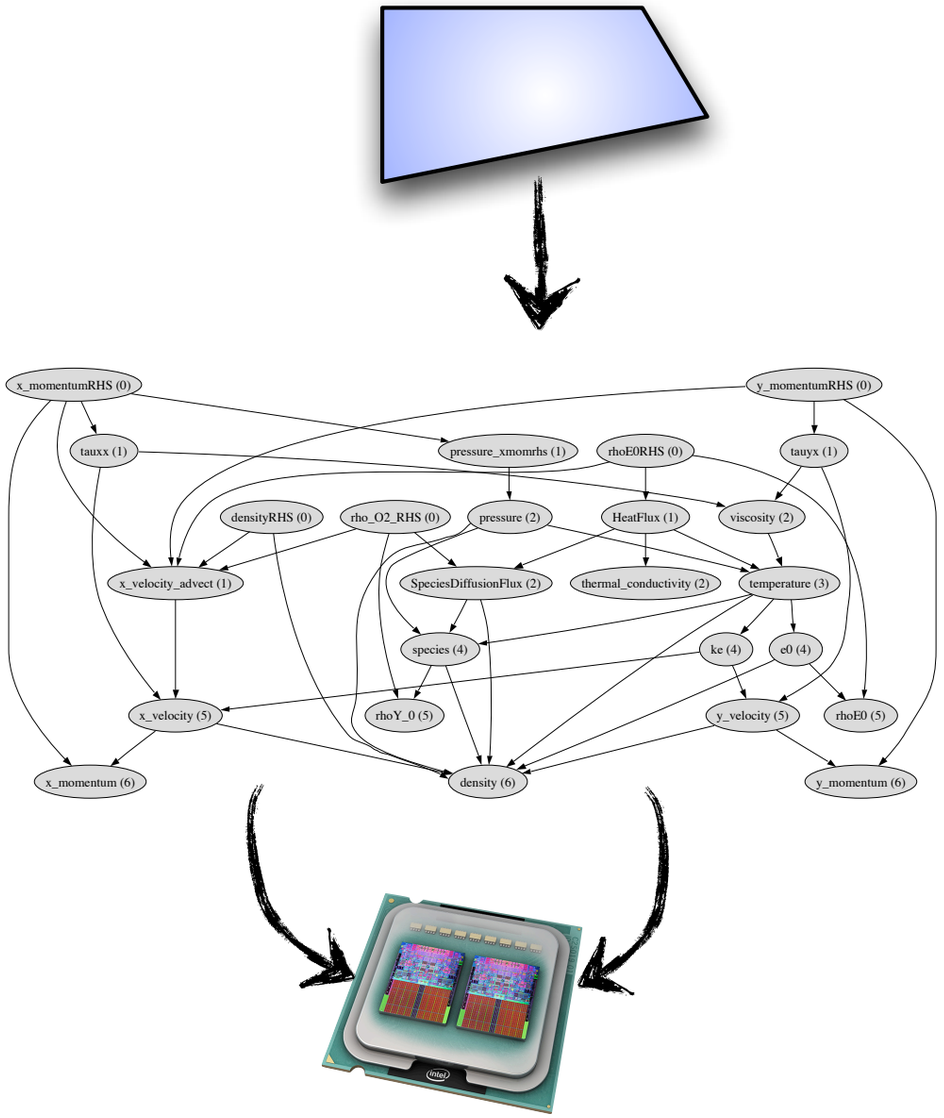
Hierarchical Parallelization

- Domain decomposition
  - data parallel (MPI)



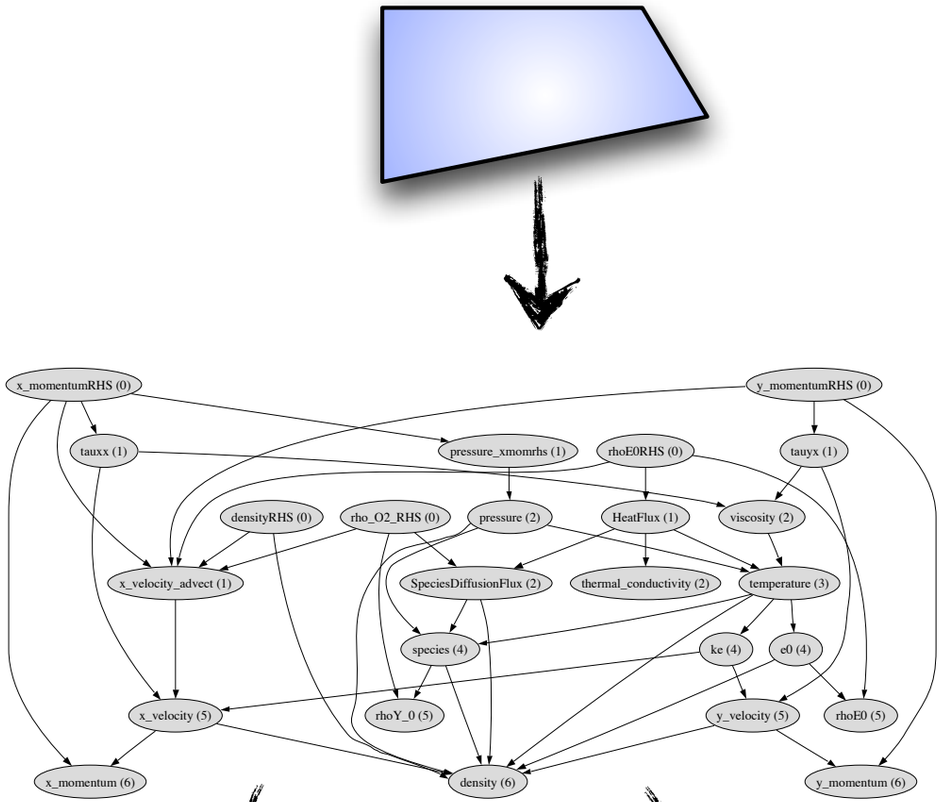
Hierarchical Parallelization

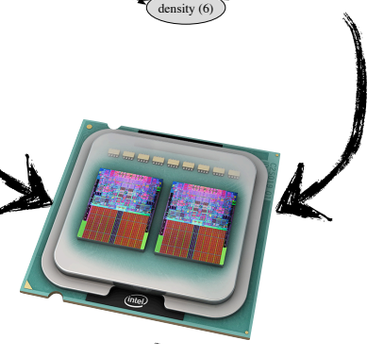
- Domain decomposition
  - data parallel (MPI)
- Algorithm decomposition
  - task parallel (threads)



Hierarchical Parallelization

- Domain decomposition
  - data parallel (MPI)
- Algorithm decomposition
  - task parallel (threads)
- Fine-grained field operations
  - data parallel (threads)



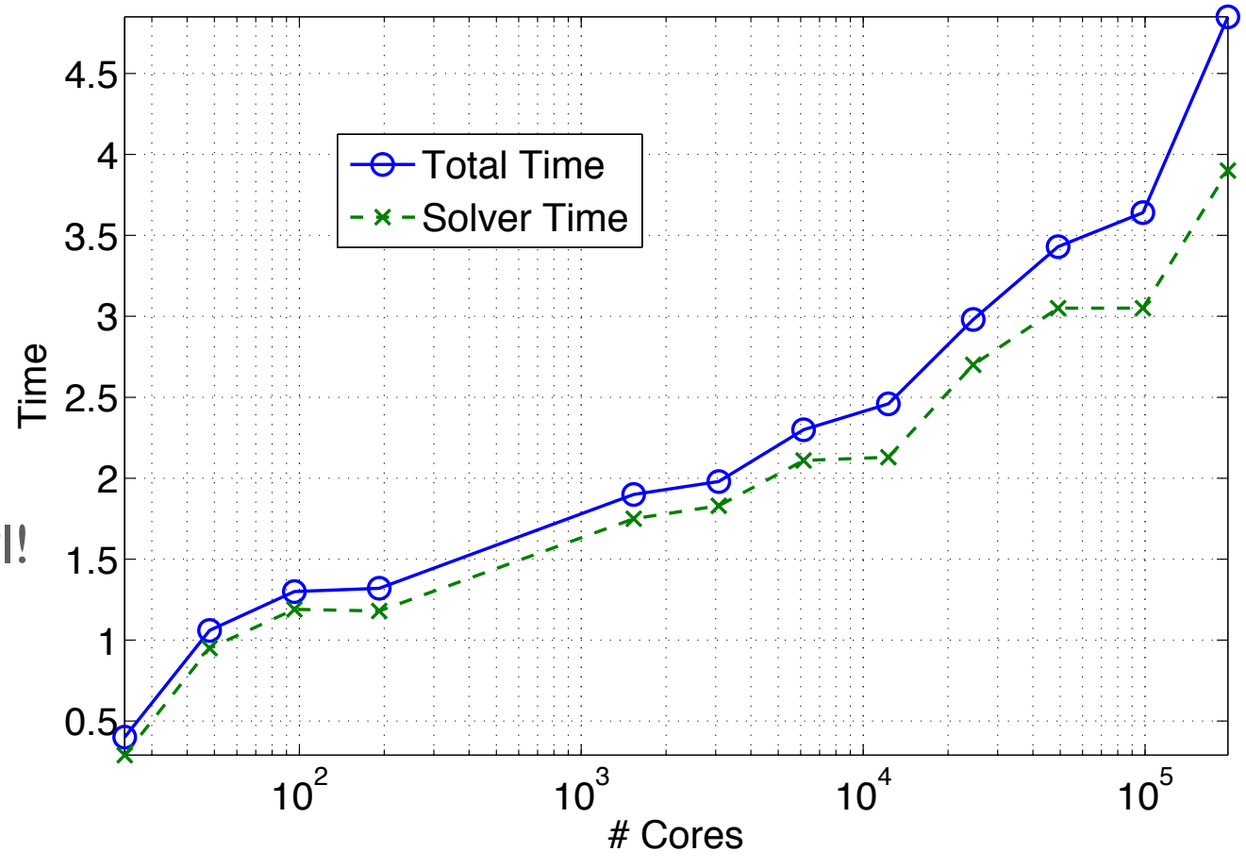


$$\nabla \cdot (\Gamma \nabla \phi) + s_\phi$$

data parallel per patch/workset

Current Scaling - Straight MPI ♻

- Efficiency:
  - MPI can scale to 200K cores (Jaguar)
    - Hypre PFMG solver, ~10 iterations required per timestep
  - Linear solver is dictating scalability
    - most of computation time is spent in linear solver
    - as we add more physics, more time will be spent outside the linear solver
- Implications:
  - Linear solvers must go multicore/GPU within MPI!
  - Consider algorithmic changes to eliminate dependency on linear solver?
- Multicore/accelerator implementations are a multiplier on MPI scalability.



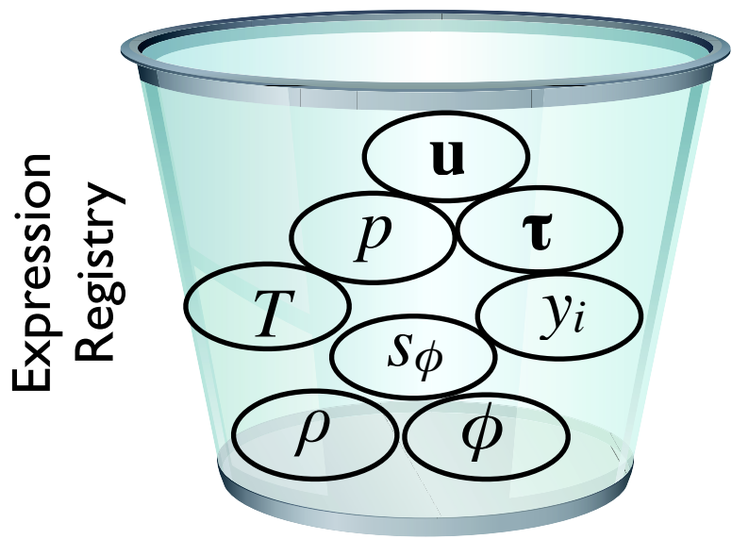
♻ Collaborative effort with Martin Berzins, funded by NSF awards OCI 0721659, PetaApps 0905068.

Task-Based Algorithm Decomposition

A Simple Example

- Register all expressions.
  - Each "expression" calculates one or more field quantities, and this is indicated to the registry.
  - Each expression advertises its *direct* dependencies

$$\frac{\partial \phi}{\partial t} = -\nabla \cdot (\phi \mathbf{u}) + \nabla \cdot (\Gamma \nabla \phi) + s_\phi$$



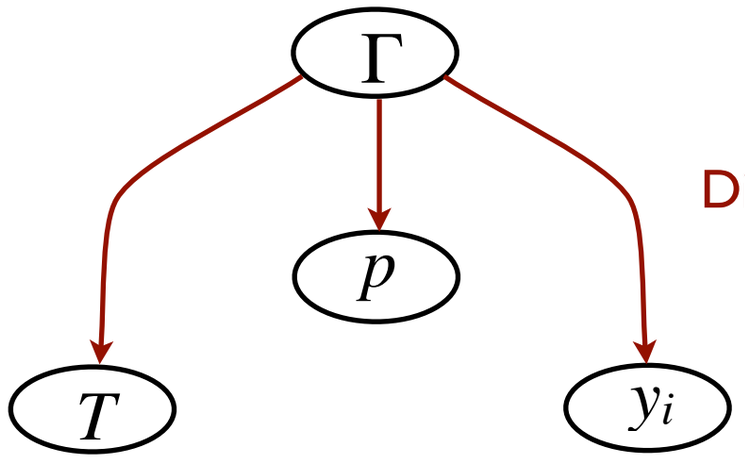
Task-Based Algorithm Decomposition

A Simple Example

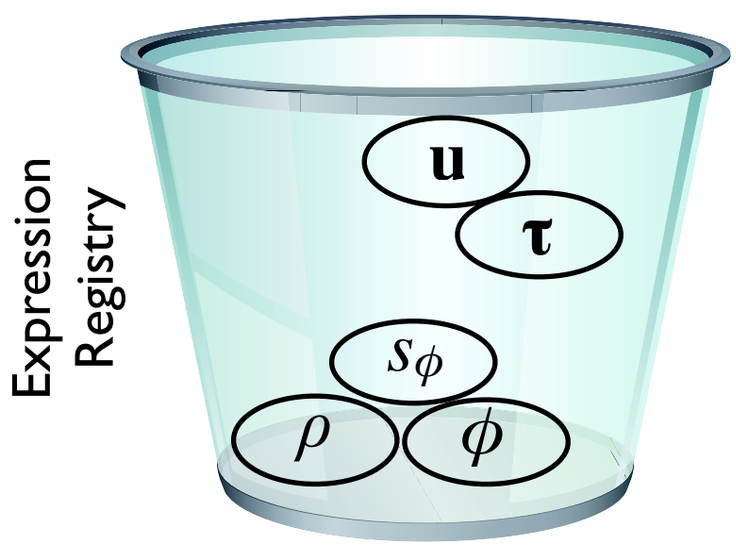
- Register all expressions.
  - Each "expression" calculates one or more field quantities, and this is indicated to the registry.
  - Each expression advertises its *direct* dependencies
- Determine a "root" expression & construct a graph.
  - All dependencies are discovered/resolved automatically and an algorithm is deduced.
- Highly localized influence of changes in models. Programmer need not know the "whole code" to make a local change.
- Not all expressions in the registry may be relevant/ used.

$$\frac{\partial \phi}{\partial t} = -\nabla \cdot (\phi \mathbf{u}) + \nabla \cdot (\Gamma \nabla \phi) + s_\phi$$

$$\Gamma = \Gamma(T, p, y_i)$$



Direct (expressed) dependencies.



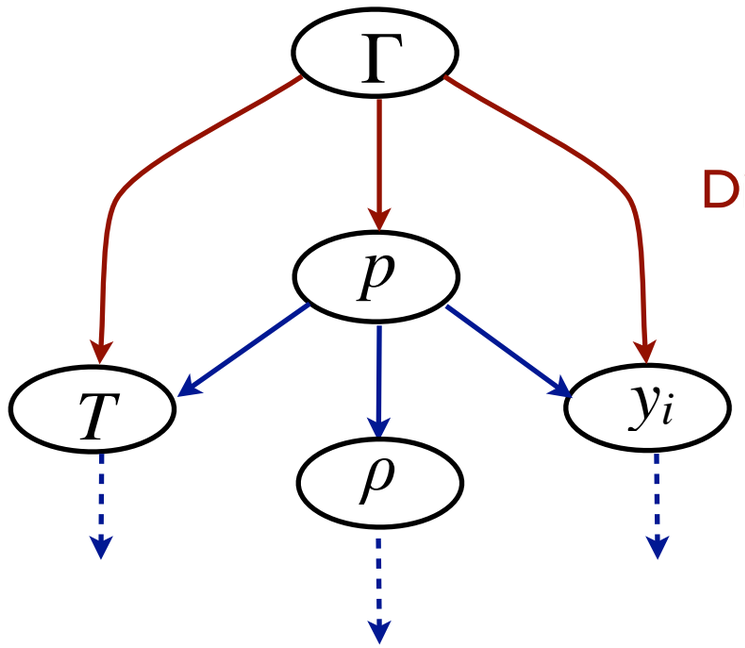
Task-Based Algorithm Decomposition

A Simple Example

- Register all expressions.
  - Each "expression" calculates one or more field quantities, and this is indicated to the registry.
  - Each expression advertises its *direct* dependencies
- Determine a "root" expression & construct a graph.
  - All dependencies are discovered/resolved automatically and an algorithm is deduced.
  - Highly localized influence of changes in models. Programmer need not know the "whole code" to make a local change.
  - Not all expressions in the registry may be relevant/ used.

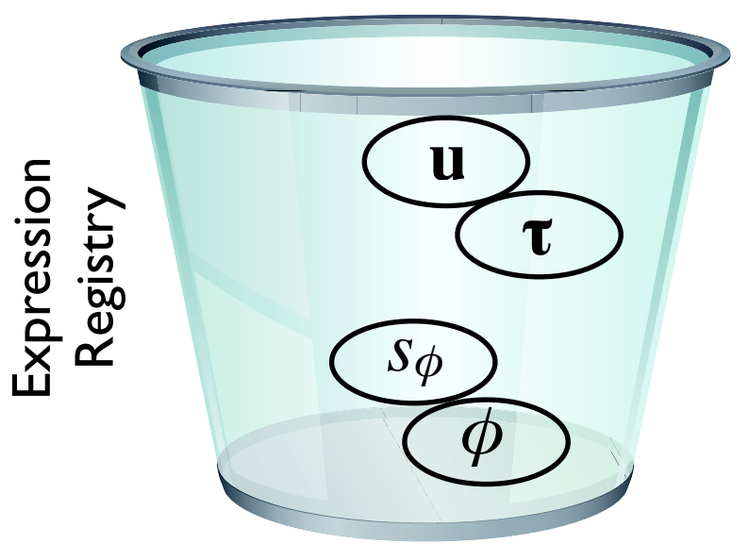
$$\frac{\partial \phi}{\partial t} = -\nabla \cdot (\phi \mathbf{u}) + \nabla \cdot (\Gamma \nabla \phi) + s_\phi$$

$$\Gamma = \Gamma(T, p, y_i)$$



Direct (expressed) dependencies.

Indirect (discovered) dependencies.



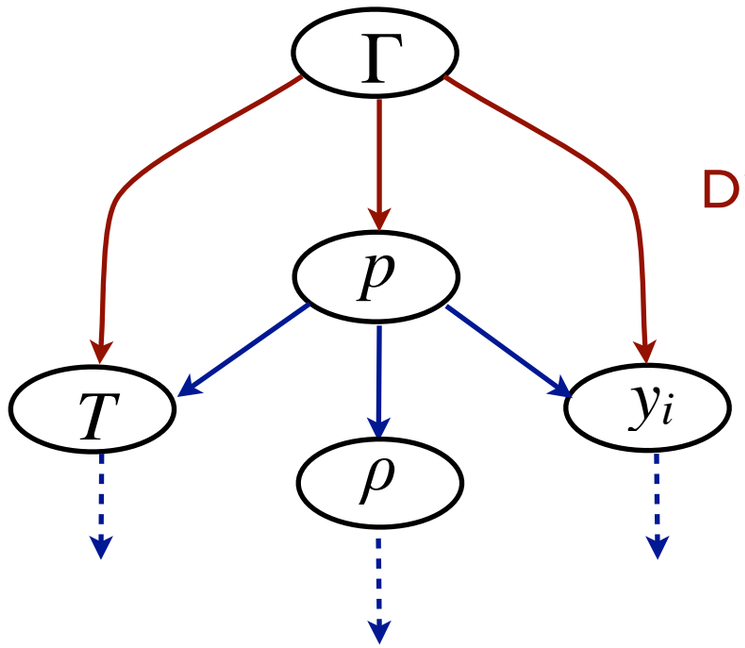
Task-Based Algorithm Decomposition

A Simple Example

- Register all expressions.
  - Each "expression" calculates one or more field quantities, and this is indicated to the registry.
  - Each expression advertises its *direct* dependencies
- Determine a "root" expression & construct a graph.
  - All dependencies are discovered/resolved automatically and an algorithm is deduced.
  - Highly localized influence of changes in models. Programmer need not know the "whole code" to make a local change.
  - Not all expressions in the registry may be relevant/ used.
- From the graph,
  - deduce storage requirements & allocate memory (externally to each expression).
  - iterate graph and bind memory.
  - schedule evaluation, ensuring proper ordering.

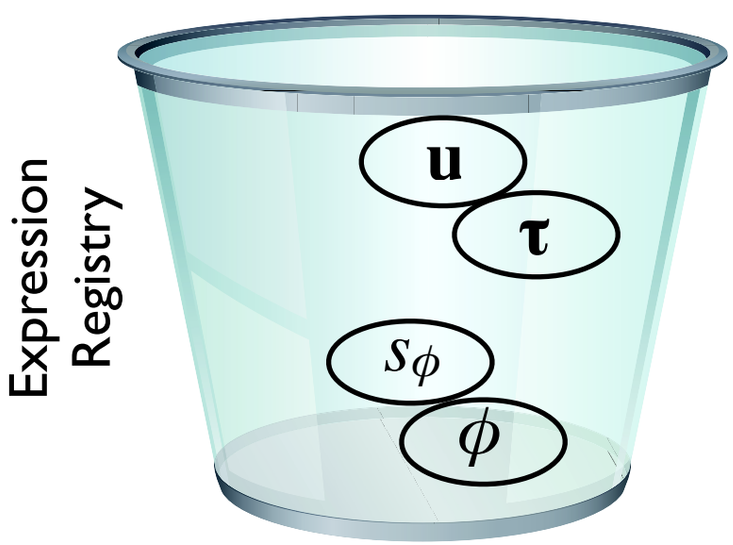
$$\frac{\partial \phi}{\partial t} = -\nabla \cdot (\phi \mathbf{u}) + \nabla \cdot (\Gamma \nabla \phi) + s_\phi$$

$$\Gamma = \Gamma(T, p, y_i)$$



Direct (expressed) dependencies.

Indirect (discovered) dependencies.



Additional Model Complexity

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \vec{u})$$
$$\frac{\partial \rho \vec{u}}{\partial t} = -\nabla \cdot (\rho \vec{u}\vec{u}) - \nabla \cdot \tau - \nabla p + \rho \vec{g}$$
$$\frac{\partial \rho e_0}{\partial t} = -\nabla \cdot (\rho e_0 \vec{u}) - \nabla \cdot (p\vec{u}) - \nabla \cdot (\tau \vec{v}) - \nabla \cdot q$$
$$\frac{\partial \rho y_i}{\partial t} = -\nabla \cdot (\rho y_i \vec{u}) - \nabla \cdot \vec{J}_i + s_i$$

Additional Model Complexity

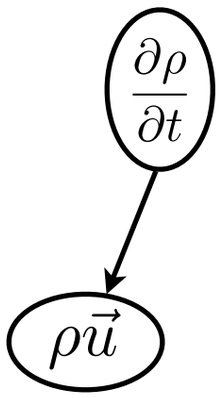
- Example: multiphase flows
  - Do we need logic in formulating each transport equation to indicate if "multiphase" is turned on?

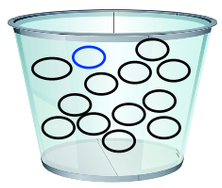
$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \vec{u}) + s_\rho$$
$$\frac{\partial \rho \vec{u}}{\partial t} = -\nabla \cdot (\rho \vec{u} \vec{u}) - \nabla \cdot \tau - \nabla p + \rho \vec{g} + s_{\rho \vec{u}}$$
$$\frac{\partial \rho e_0}{\partial t} = -\nabla \cdot (\rho e_0 \vec{u}) - \nabla \cdot (p \vec{u}) - \nabla \cdot (\tau \vec{v}) - \nabla \cdot q + + s_{\rho e_0}$$
$$\frac{\partial \rho y_i}{\partial t} = -\nabla \cdot (\rho y_i \vec{u}) - \nabla \cdot \vec{J}_i + s_i + s_{\rho y_i}$$

Additional Model Complexity

- Example: multiphase flows
  - Do we need logic in formulating each transport equation to indicate if "multiphase" is turned on?

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \vec{u}) + s_\rho$$
$$\frac{\partial \rho \vec{u}}{\partial t} = -\nabla \cdot (\rho \vec{u} \vec{u}) - \nabla \cdot \tau - \nabla p + \rho \vec{g} + s_{\rho \vec{u}}$$
$$\frac{\partial \rho e_0}{\partial t} = -\nabla \cdot (\rho e_0 \vec{u}) - \nabla \cdot (p \vec{u}) - \nabla \cdot (\tau \vec{v}) - \nabla \cdot q + + s_{\rho e_0}$$
$$\frac{\partial \rho y_i}{\partial t} = -\nabla \cdot (\rho y_i \vec{u}) - \nabla \cdot \vec{J}_i + s_i + s_{\rho y_i}$$



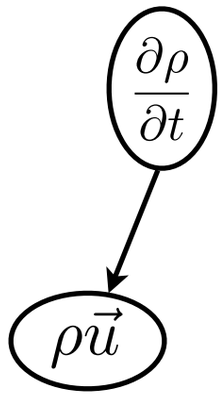


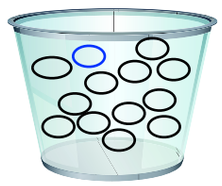
Additional Model Complexity

- Example: multiphase flows
  - Do we need logic in formulating each transport equation to indicate if "multiphase" is turned on?

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \vec{u}) + s_\rho$$
$$\frac{\partial \rho \vec{u}}{\partial t} = -\nabla \cdot (\rho \vec{u} \vec{u}) - \nabla \cdot \tau - \nabla p + \rho \vec{g} + s_{\rho \vec{u}}$$
$$\frac{\partial \rho e_0}{\partial t} = -\nabla \cdot (\rho e_0 \vec{u}) - \nabla \cdot (p \vec{u}) - \nabla \cdot (\tau \vec{v}) - \nabla \cdot q + + s_{\rho e_0}$$
$$\frac{\partial \rho y_i}{\partial t} = -\nabla \cdot (\rho y_i \vec{u}) - \nabla \cdot \vec{J}_i + s_i + s_{\rho y_i}$$

- "Attach" a dependency to an expression.
  - Allows "push" coupling rather than "pull" coupling
  - Physics that knows about coupling (particle transport) injects appropriate coupling terms into gas-phase solver.
  - Full dependency structure is maintained; no "code creep" from additional models.

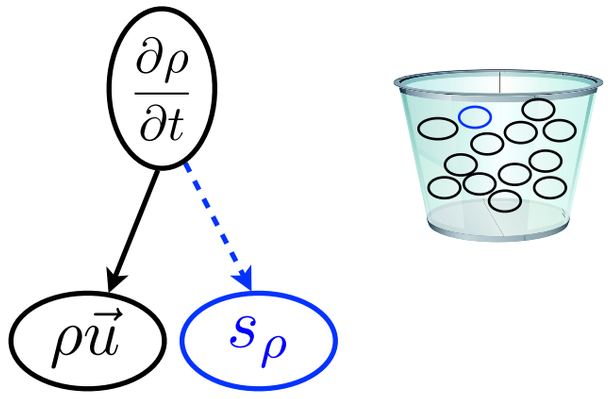




Additional Model Complexity

- Example: multiphase flows
  - Do we need logic in formulating each transport equation to indicate if "multiphase" is turned on?
- "Attach" a dependency to an expression.
  - Allows "push" coupling rather than "pull" coupling
  - Physics that knows about coupling (particle transport) injects appropriate coupling terms into gas-phase solver.
  - Full dependency structure is maintained; no "code creep" from additional models.

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \vec{u}) + s_\rho$$
$$\frac{\partial \rho \vec{u}}{\partial t} = -\nabla \cdot (\rho \vec{u} \vec{u}) - \nabla \cdot \tau - \nabla p + \rho \vec{g} + s_{\rho \vec{u}}$$
$$\frac{\partial \rho e_0}{\partial t} = -\nabla \cdot (\rho e_0 \vec{u}) - \nabla \cdot (p \vec{u}) - \nabla \cdot (\tau \vec{v}) - \nabla \cdot q + + s_{\rho e_0}$$
$$\frac{\partial \rho y_i}{\partial t} = -\nabla \cdot (\rho y_i \vec{u}) - \nabla \cdot \vec{J}_i + s_i + s_{\rho y_i}$$



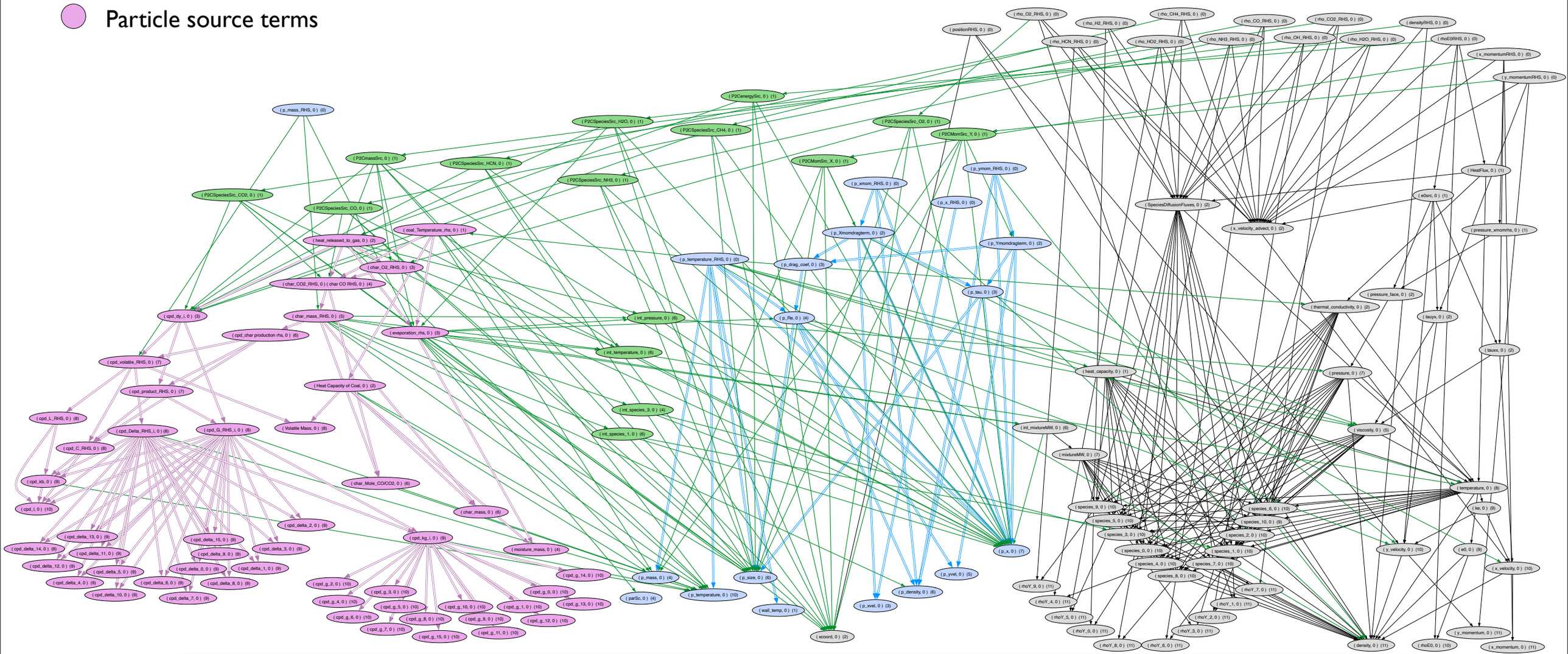

```
registry->attach_dependency( rhoRHS, srho, ADD );
```

$\partial \varrho / \partial t$ doesn't directly advertise a dependency on s_ϱ.

Example Graph - Coal Combustion/Gasification

- Gas phase terms (CO/H2 combustion - moving to methane...)
- Particle evolution terms
- Particle - gas coupling terms
- Particle source terms

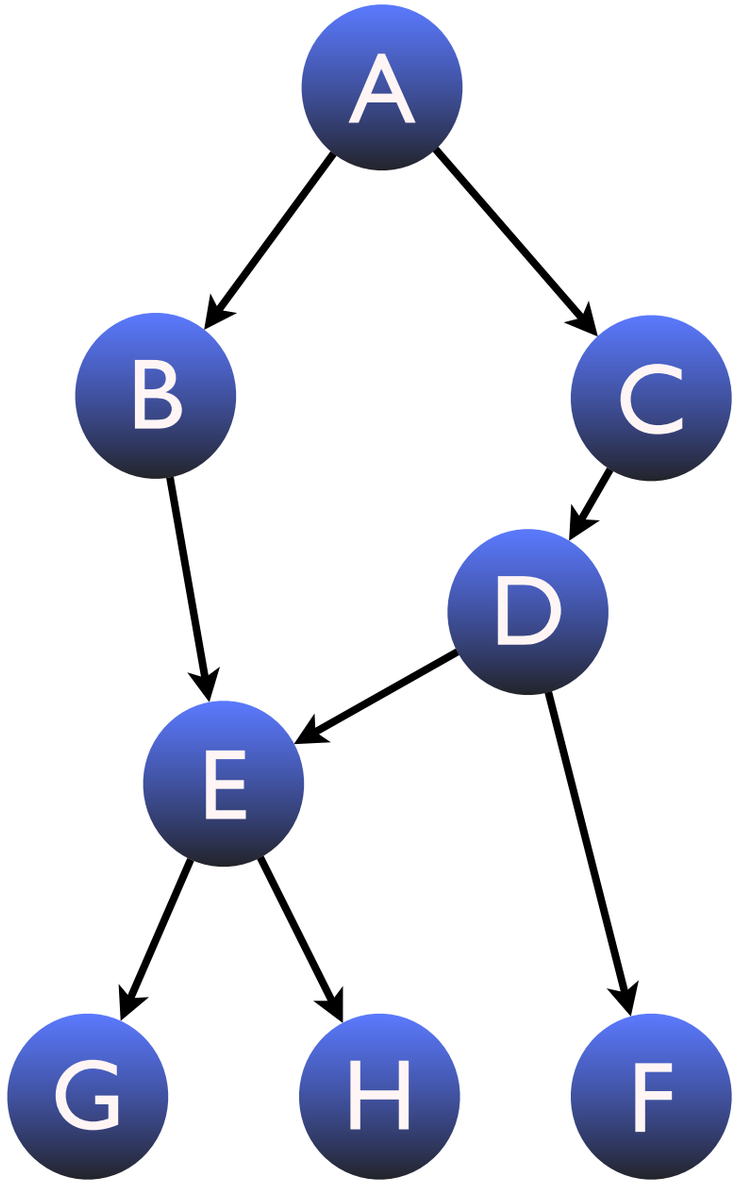
15 Gas-phase PDEs,
~25 ODEs per particle.



NO modification to gas-phase code to get 2-way coupling on mass, momentum, energy!

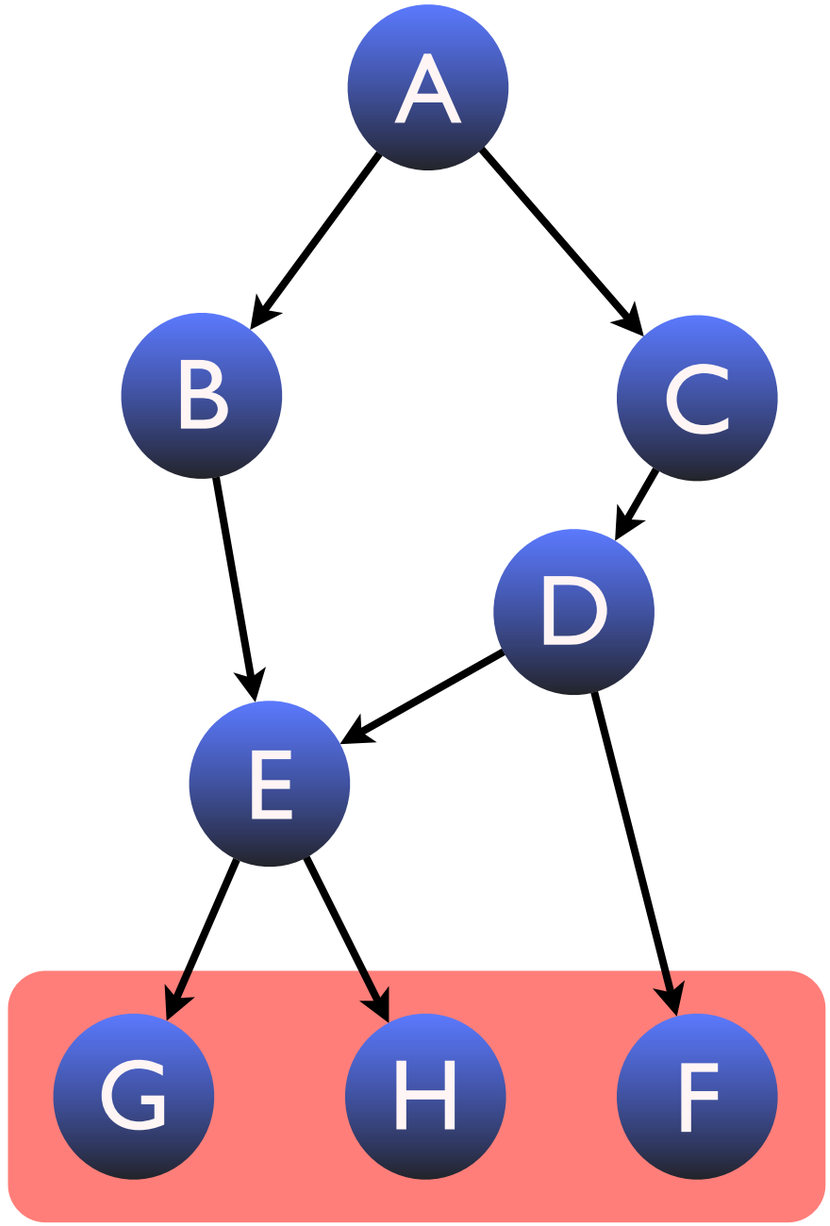
Multicore Scheduling

- "Bottom" nodes are placed in execution queue.
- When a node completes, its "wait count" is decremented.
- When all of a node's "children" are done (wait count=0) it is placed in the priority execution queue.
  - priority determined to optimize graph execution time.
  - backfilling naturally occurs (particularly for broad graphs) while "heavy" nodes execute.
  - resources may be dynamically migrated between thread pool associated with task graph to threads associated with data-parallel execution (within a node)



Multicore Scheduling

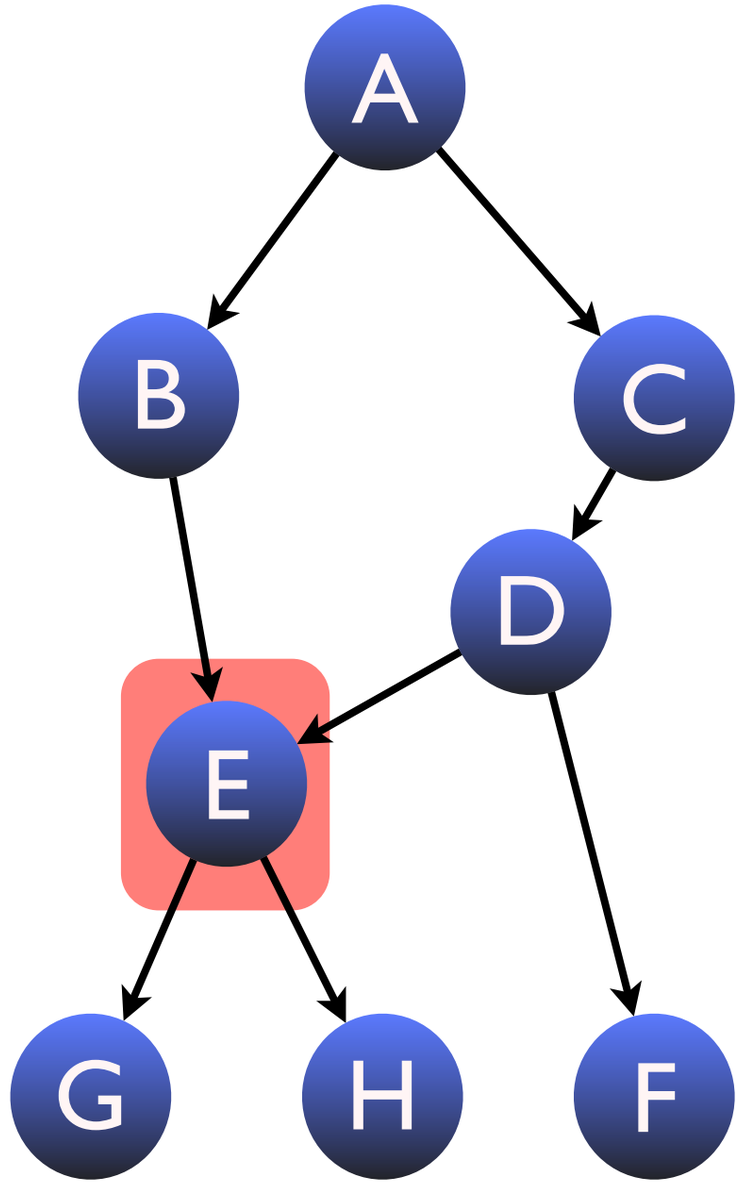
- "Bottom" nodes are placed in execution queue.
- When a node completes, its "wait count" is decremented.
- When all of a node's "children" are done (wait count=0) it is placed in the priority execution queue.
  - priority determined to optimize graph execution time.
  - backfilling naturally occurs (particularly for broad graphs) while "heavy" nodes execute.
  - resources may be dynamically migrated between thread pool associated with task graph to threads associated with data-parallel execution (within a node)



Prioritize G, H because they are "deeper" than F.

Multicore Scheduling

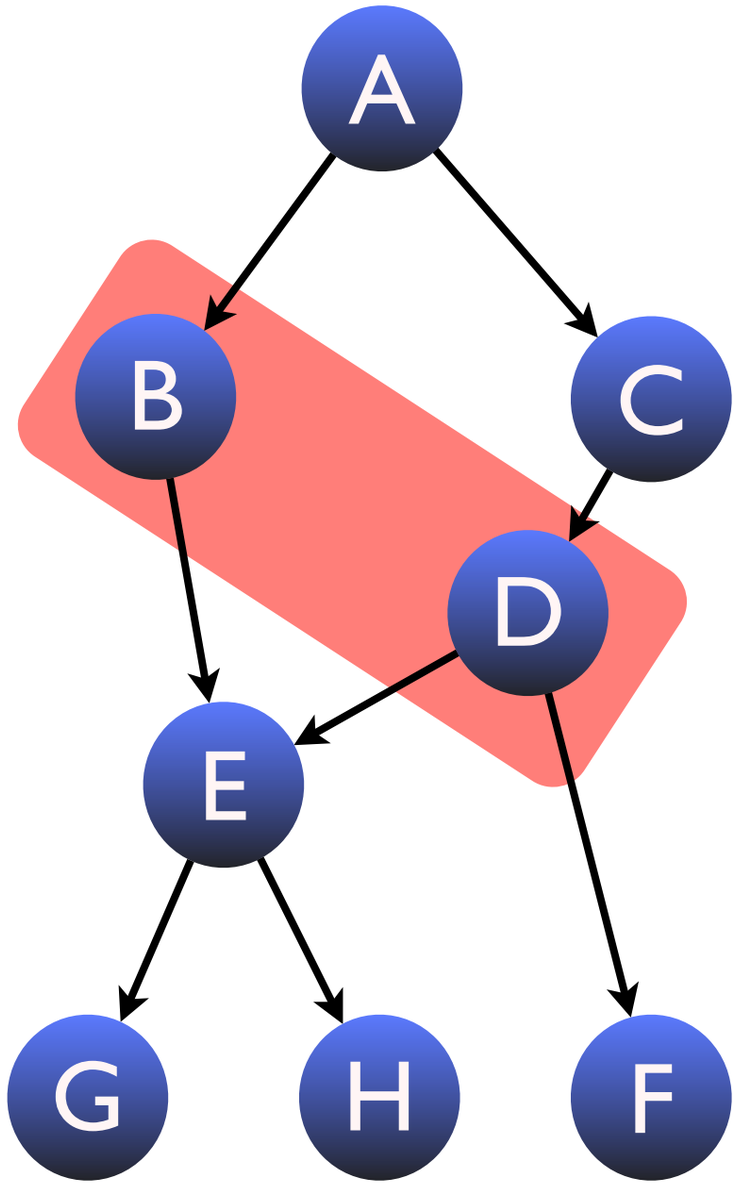
- "Bottom" nodes are placed in execution queue.
- When a node completes, its "wait count" is decremented.
- When all of a node's "children" are done (wait count=0) it is placed in the priority execution queue.
  - priority determined to optimize graph execution time.
  - backfilling naturally occurs (particularly for broad graphs) while "heavy" nodes execute.
  - resources may be dynamically migrated between thread pool associated with task graph to threads associated with data-parallel execution (within a node)



Serialization point - push resources into data parallel on E.

Multicore Scheduling

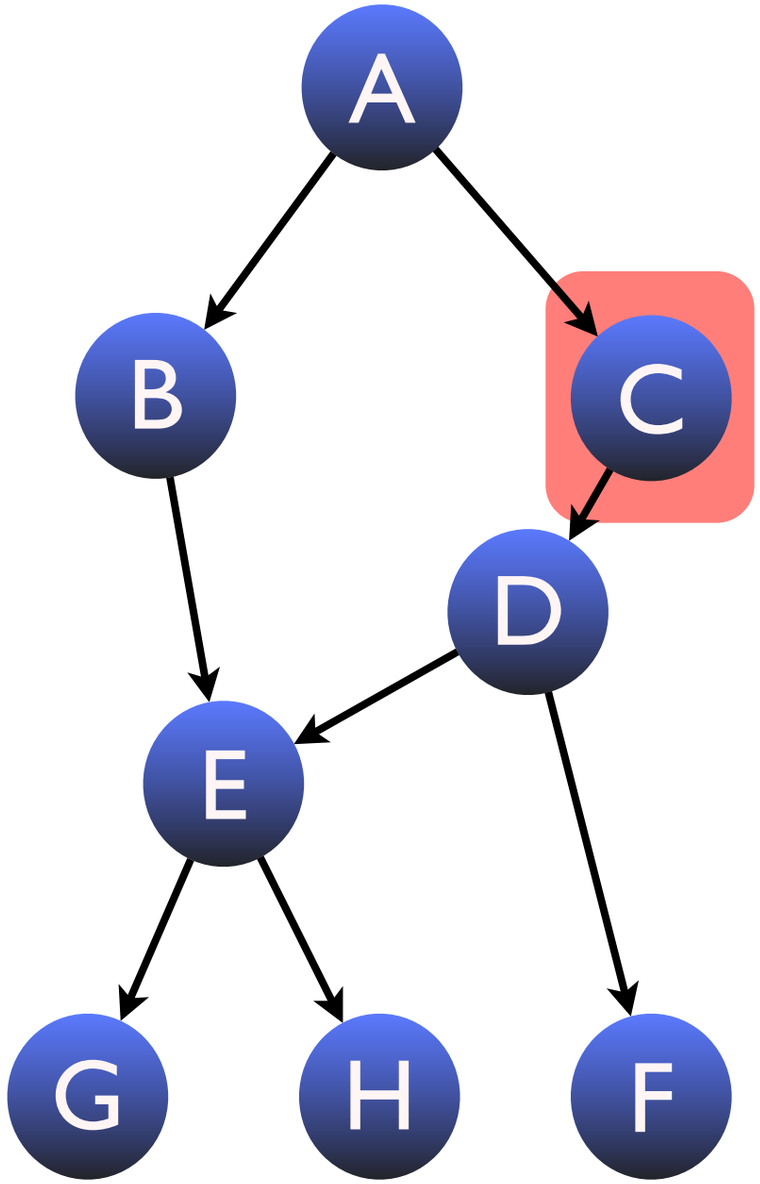
- "Bottom" nodes are placed in execution queue.
- When a node completes, its "wait count" is decremented.
- When all of a node's "children" are done (wait count=0) it is placed in the priority execution queue.
  - priority determined to optimize graph execution time.
  - backfilling naturally occurs (particularly for broad graphs) while "heavy" nodes execute.
  - resources may be dynamically migrated between thread pool associated with task graph to threads associated with data-parallel execution (within a node)



Prioritize D (may push resources into "D" if possible)

Multicore Scheduling

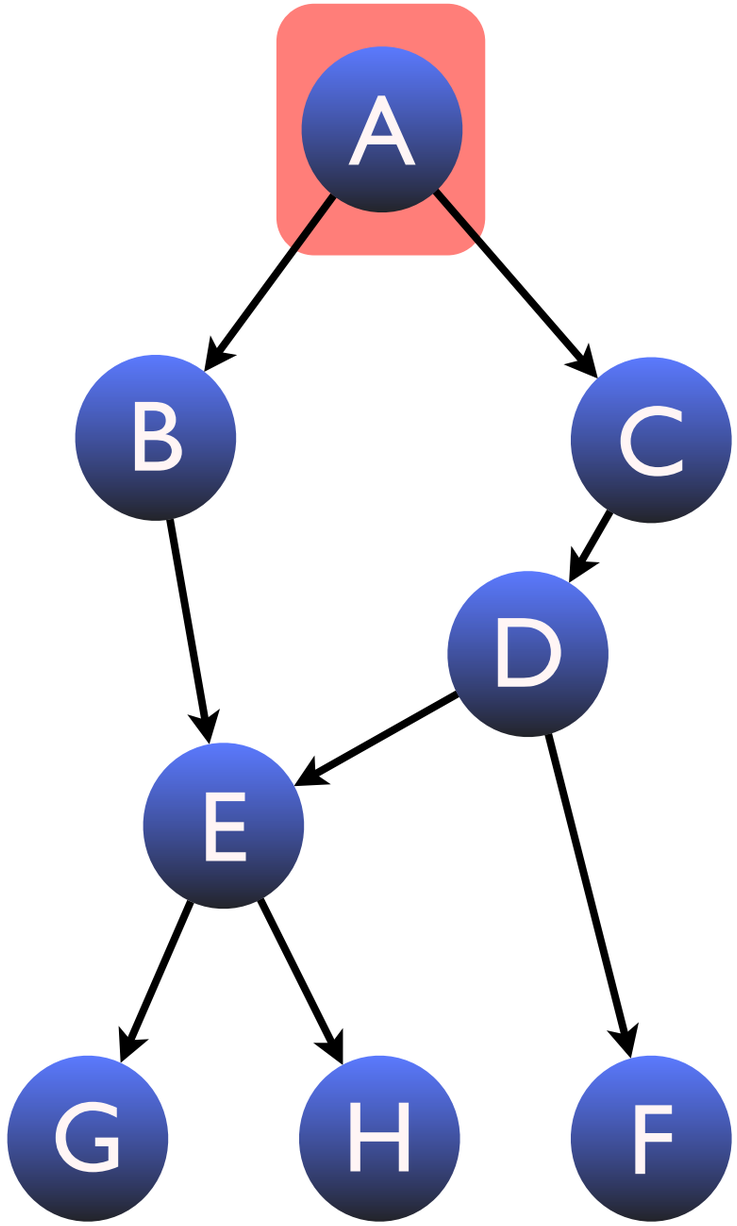
- "Bottom" nodes are placed in execution queue.
- When a node completes, its "wait count" is decremented.
- When all of a node's "children" are done (wait count=0) it is placed in the priority execution queue.
  - priority determined to optimize graph execution time.
  - backfilling naturally occurs (particularly for broad graphs) while "heavy" nodes execute.
  - resources may be dynamically migrated between thread pool associated with task graph to threads associated with data-parallel execution (within a node)



Serialization point - push resources into data parallel on C.

Multicore Scheduling

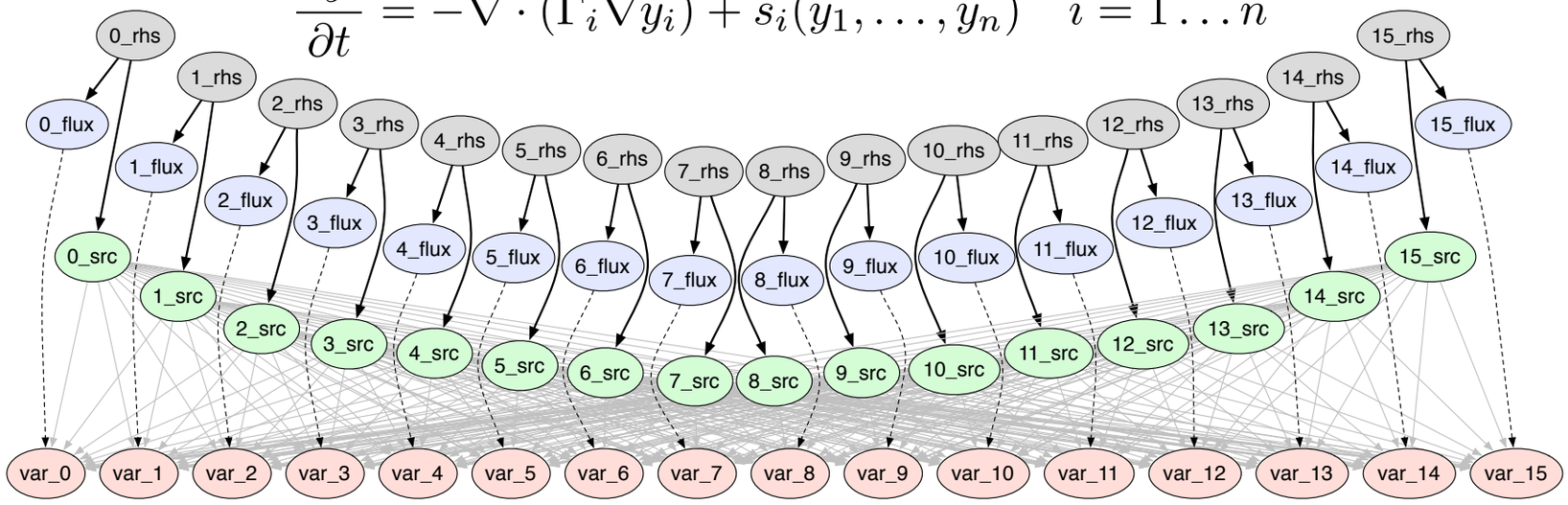
- "Bottom" nodes are placed in execution queue.
- When a node completes, its "wait count" is decremented.
- When all of a node's "children" are done (wait count=0) it is placed in the priority execution queue.
  - priority determined to optimize graph execution time.
  - backfilling naturally occurs (particularly for broad graphs) while "heavy" nodes execute.
  - resources may be dynamically migrated between thread pool associated with task graph to threads associated with data-parallel execution (within a node)



Serialization point - push resources into data parallel on A.

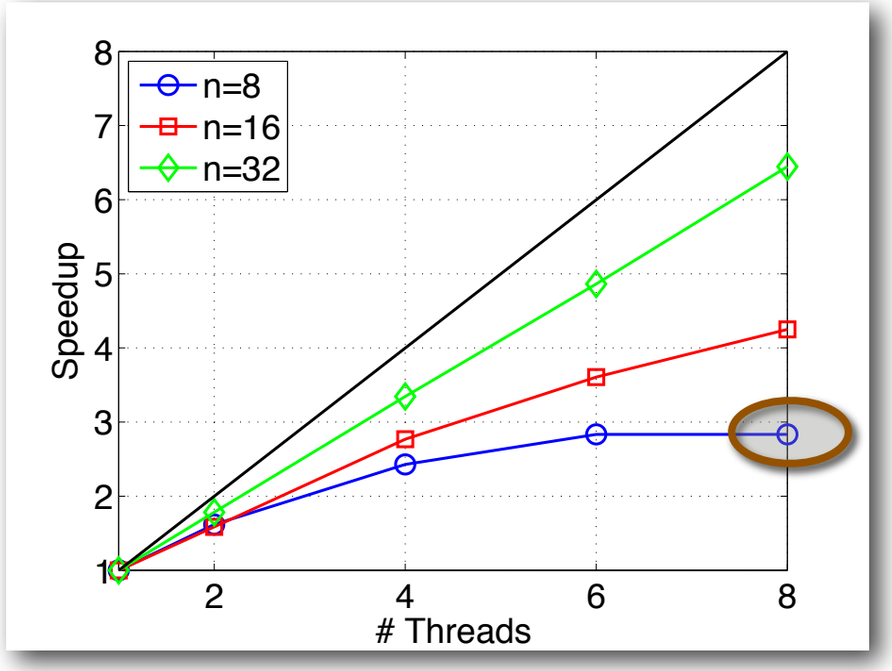
Idealized Scaling Example (Task Parallel)

$$\frac{\partial y_i}{\partial t} = -\nabla \cdot (\Gamma_i \nabla y_i) + s_i(y_1, \ldots, y_n) \quad i = 1 \ldots n$$



Graph analysis indicates theoretical parallelizability and speedup.

n	$\mathcal{P}$	$\mathcal{S}_\infty$
8	0.89	7.5
16	0.94	13.7
32	0.97	28.6



- Broad graphs required for task-based parallelism.
- Currently not obtaining close to theoretical scalability - cache issues?
- To obtain many-core scalability, we need to expose concurrency within each task!

P. K. Notz, R. P. Pawlowski, and J. C. Sutherland, "Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software," *ACM Transactions on Mathematical Software*, to appear, 2012

Granularity & Multicore Scaling (Data Parallel)

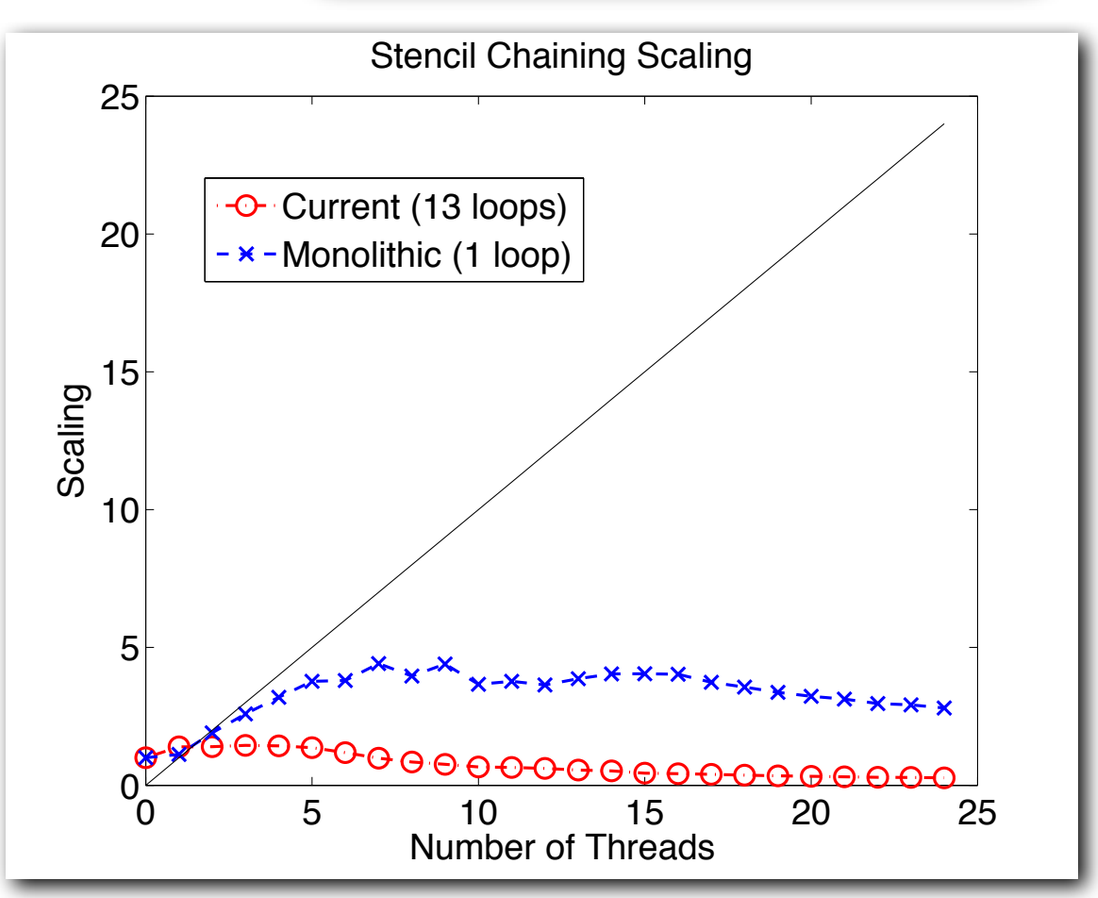
$$\frac{\partial \phi}{\partial t} = -\frac{\partial}{\partial x}\left(-\lambda \frac{\partial \phi}{\partial x}\right) - \frac{\partial}{\partial y}\left(-\lambda \frac{\partial \phi}{\partial y}\right) + \frac{\partial}{\partial z}\left(-\lambda \frac{\partial \phi}{\partial z}\right)$$

"Naive" implementation

1. Calculate λ
2. Interpolate λ to faces (3 loops)
3. Calculate gradient of ϕ at faces (3 loops)
4. Multiply $\partial\phi/\partial x_i$ by λ (3 loops)
5. Calculate divergence (3 loops)
6. Form full RHS (1 loop)

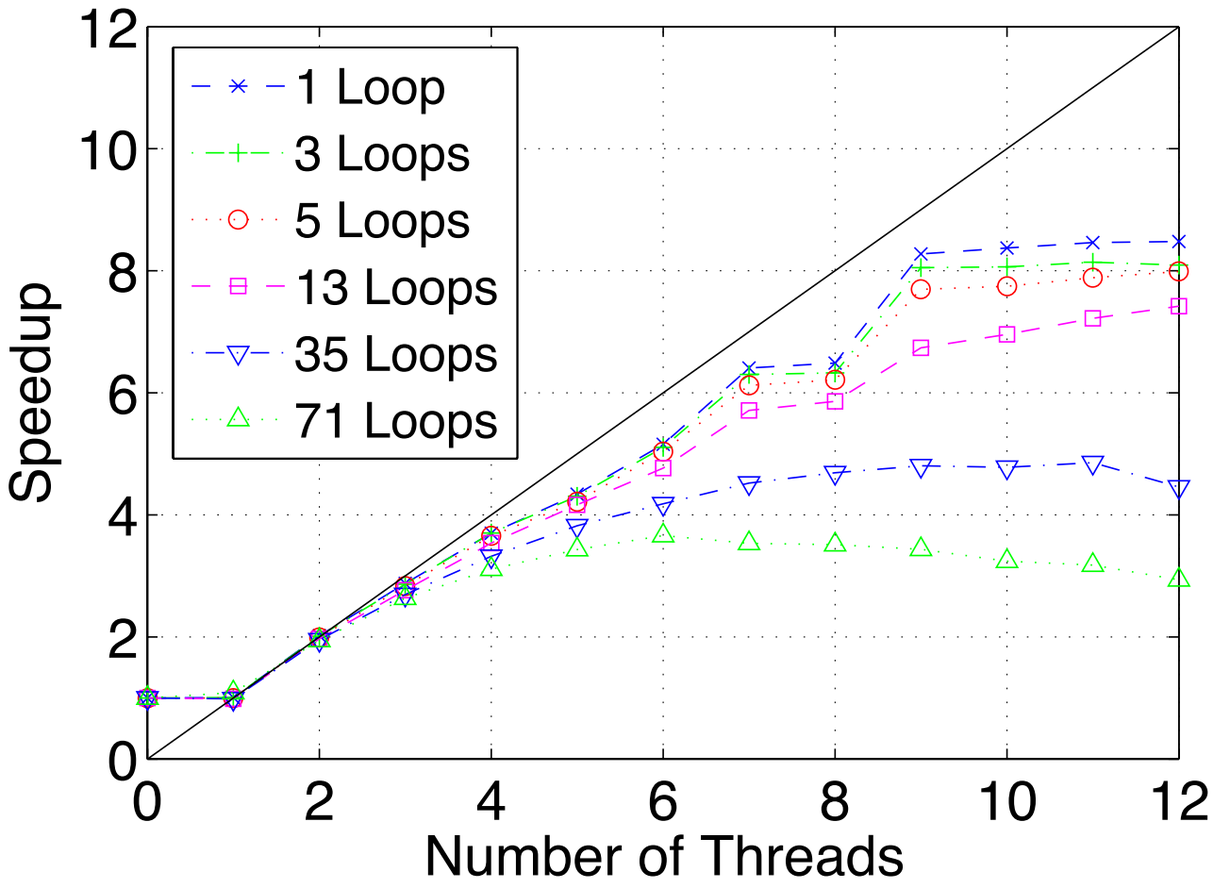
Loop Fusing

1 loop, λ and $\partial\phi/\partial x_i$ held as temporaries on inner loops.



- Cache contention is a real issue (work/write)
- Loop fusing is important (works against traditional abstraction)
- DSLs can help ease this pain significantly!

Domain-Specific Languages



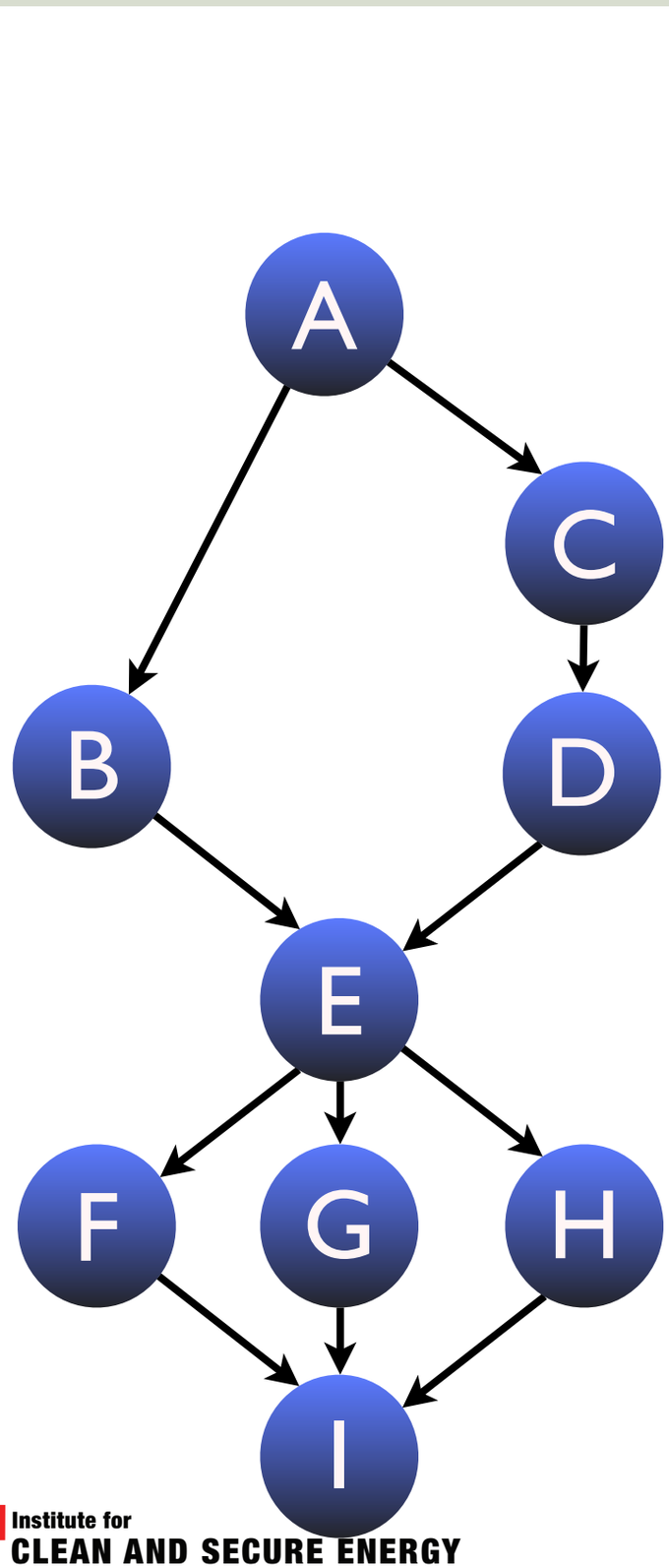
- Need "loop fusing" - more work per loop, fewer loops.
- Many threads overload memory bus.

$$\frac{\partial}{\partial x}\left(\Gamma\frac{\partial\phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(\Gamma\frac{\partial\phi}{\partial y}\right)+\frac{\partial}{\partial z}\left(\Gamma\frac{\partial\phi}{\partial z}\right)+s_\phi$$

```
rhs <<= Dx( Rx(Gamma) * Gx(phi) )
      + Dy( Ry(Gamma) * Gy(phi) )
      + Dz( Rz(Gamma) * Gz(phi) )
      + Sphi
```

DSL hides details of implementation - fast refactors

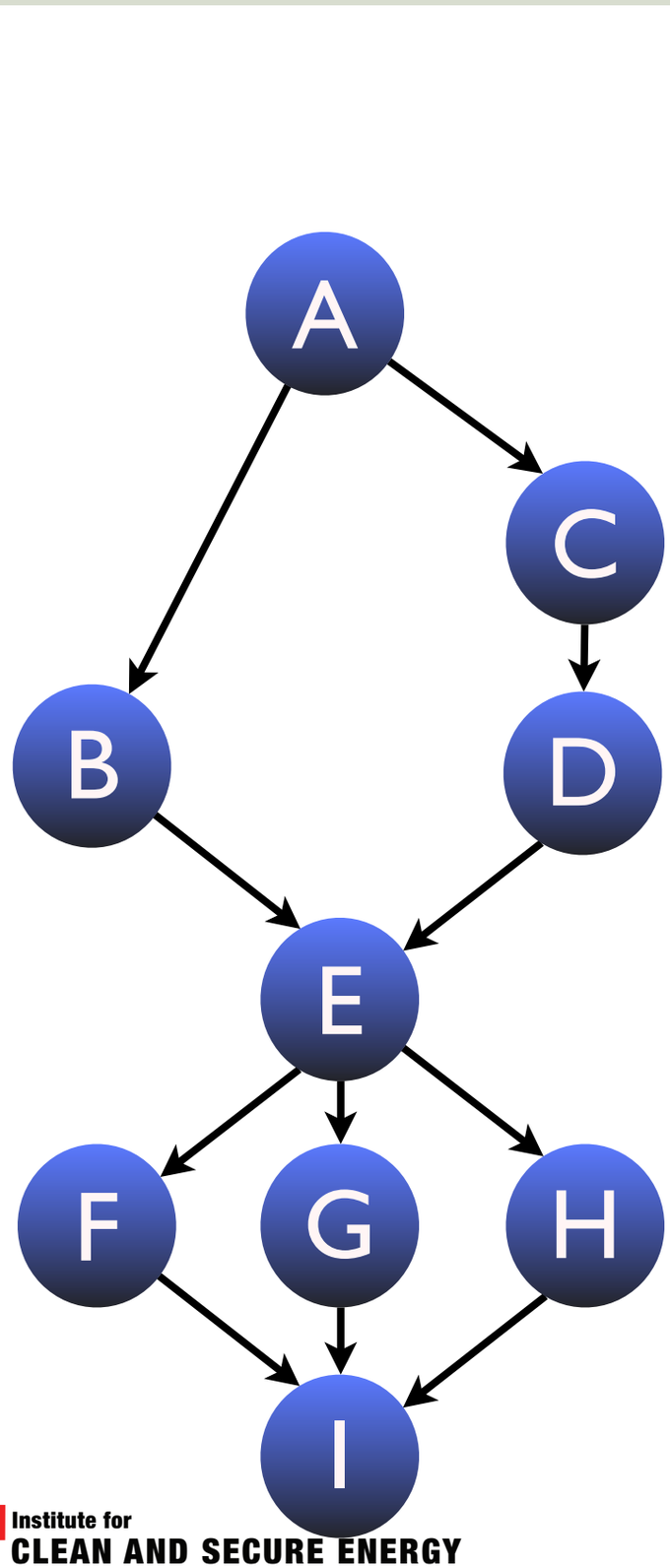
Memory Management - Pool Allocation



Static fields

Dynamic fields

Memory Management - Pool Allocation



Stage 1

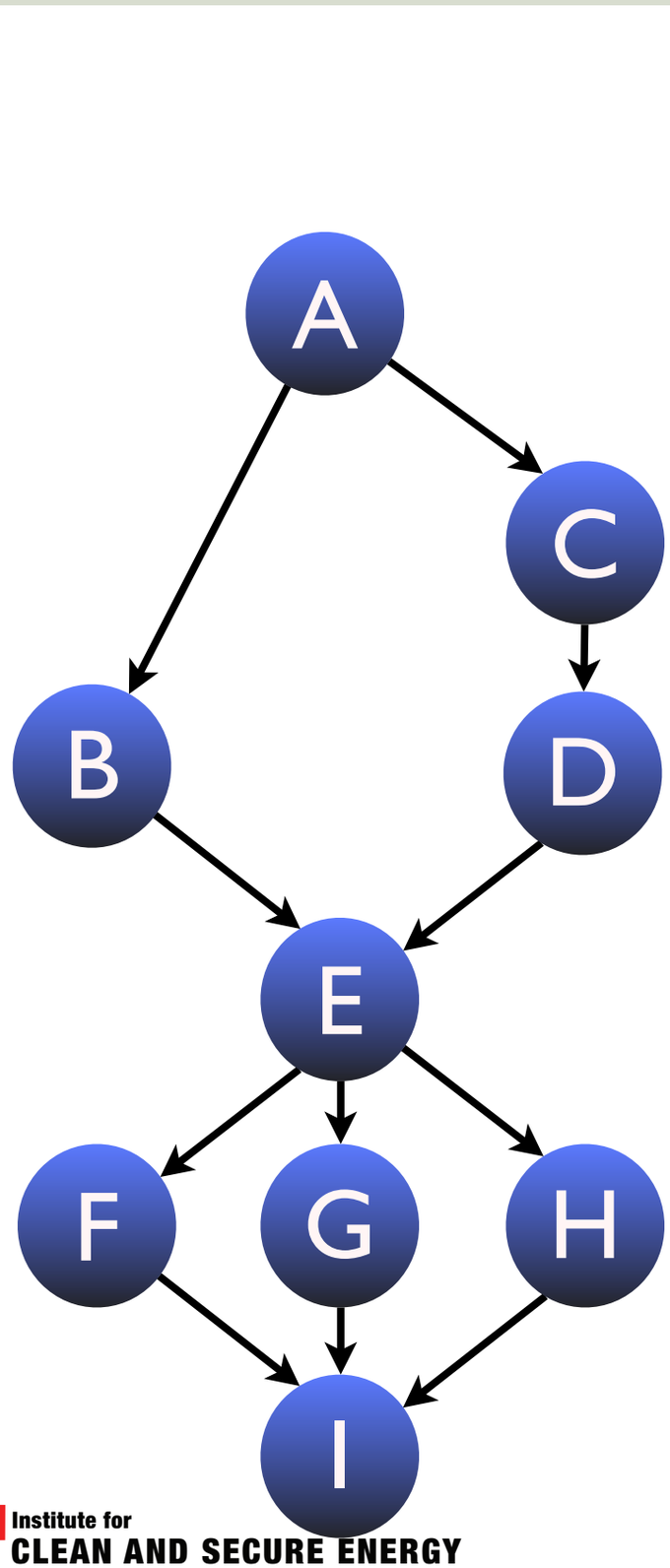
'I', 'A' allocated statically

Static fields

I A

Dynamic fields

Memory Management - Pool Allocation



Static fields

I A

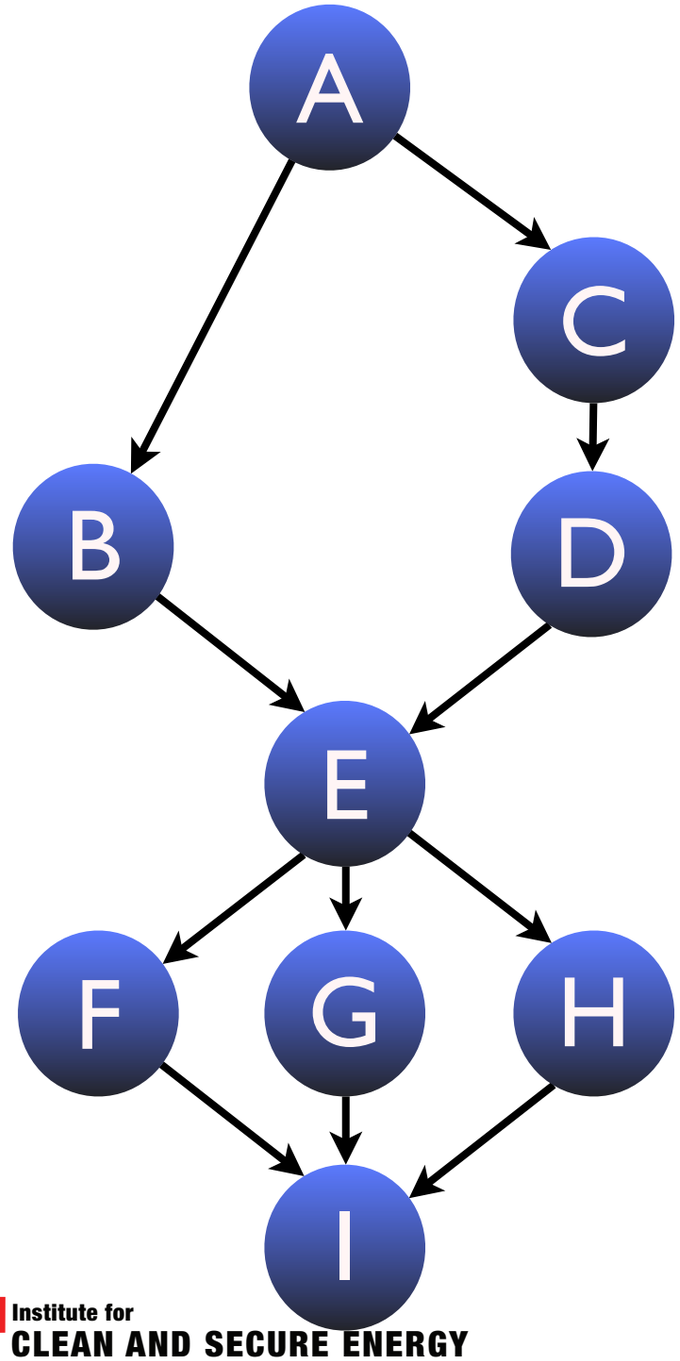
Dynamic fields

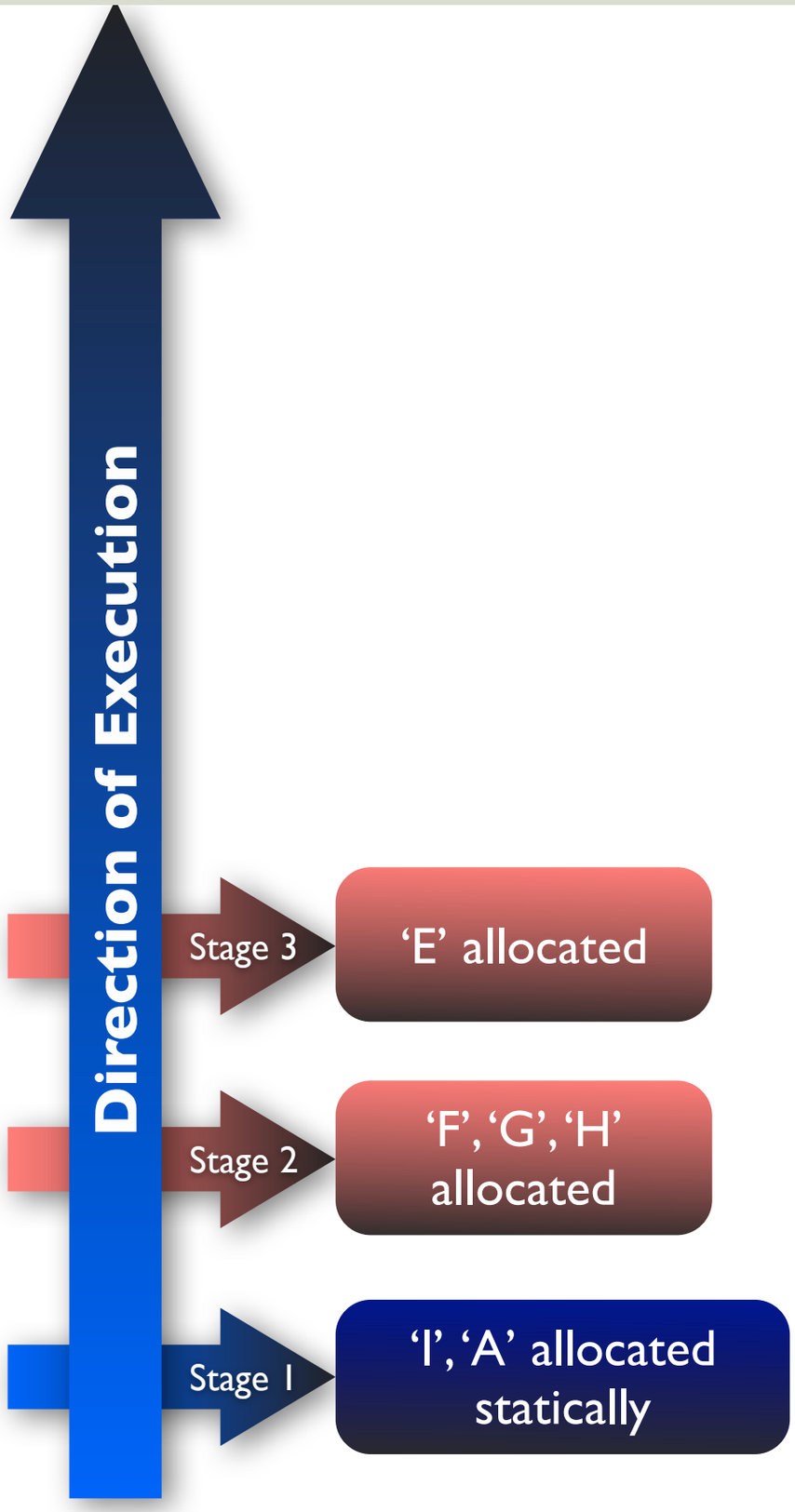
F G H

Stage 2 → 'F', 'G', 'H' allocated

Stage 1 → 'I', 'A' allocated statically

Memory Management - Pool Allocation





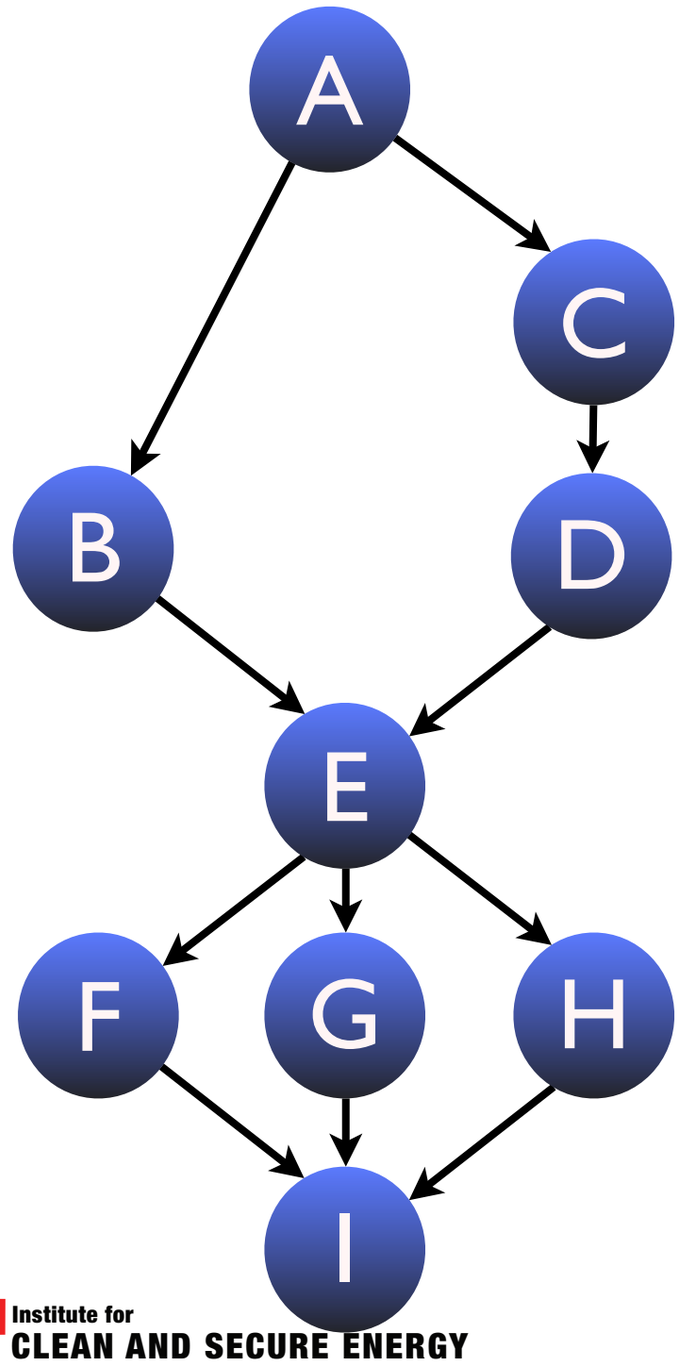
Static fields

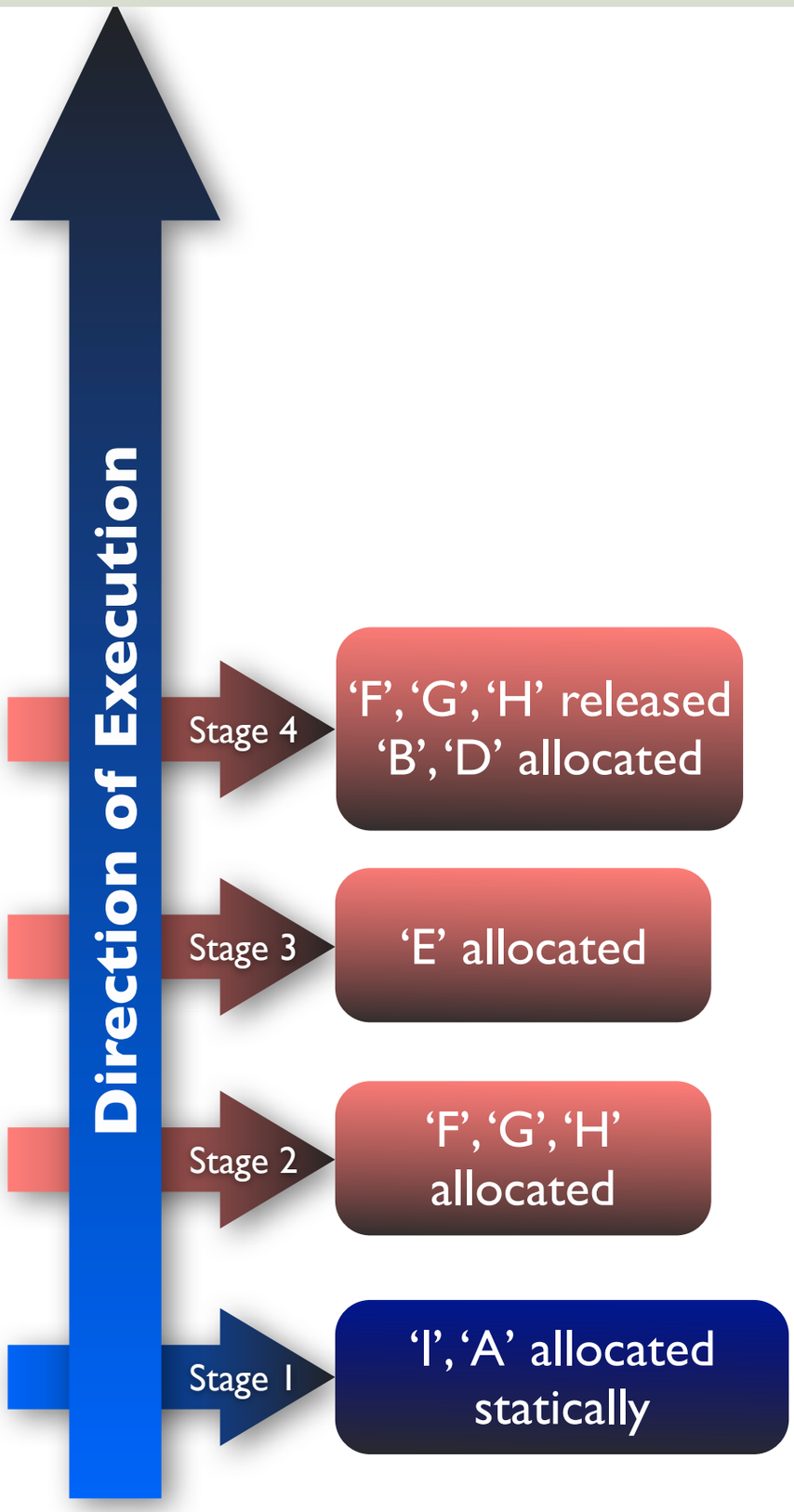
I A

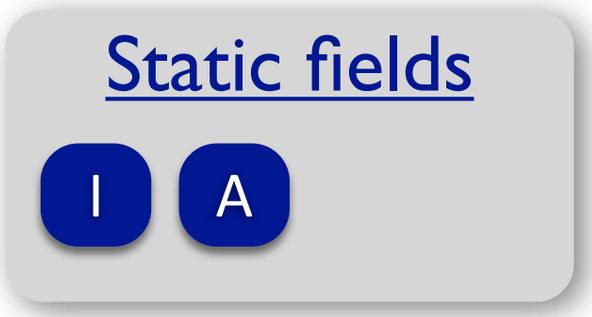
Dynamic fields

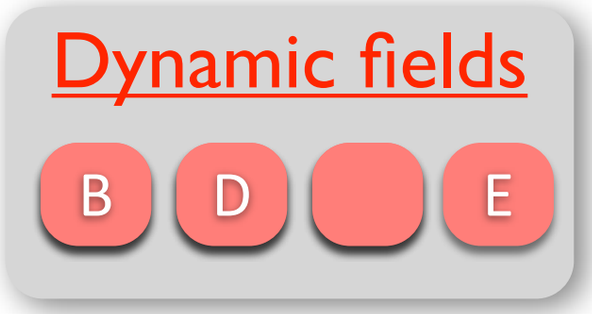
F G H E

Memory Management - Pool Allocation

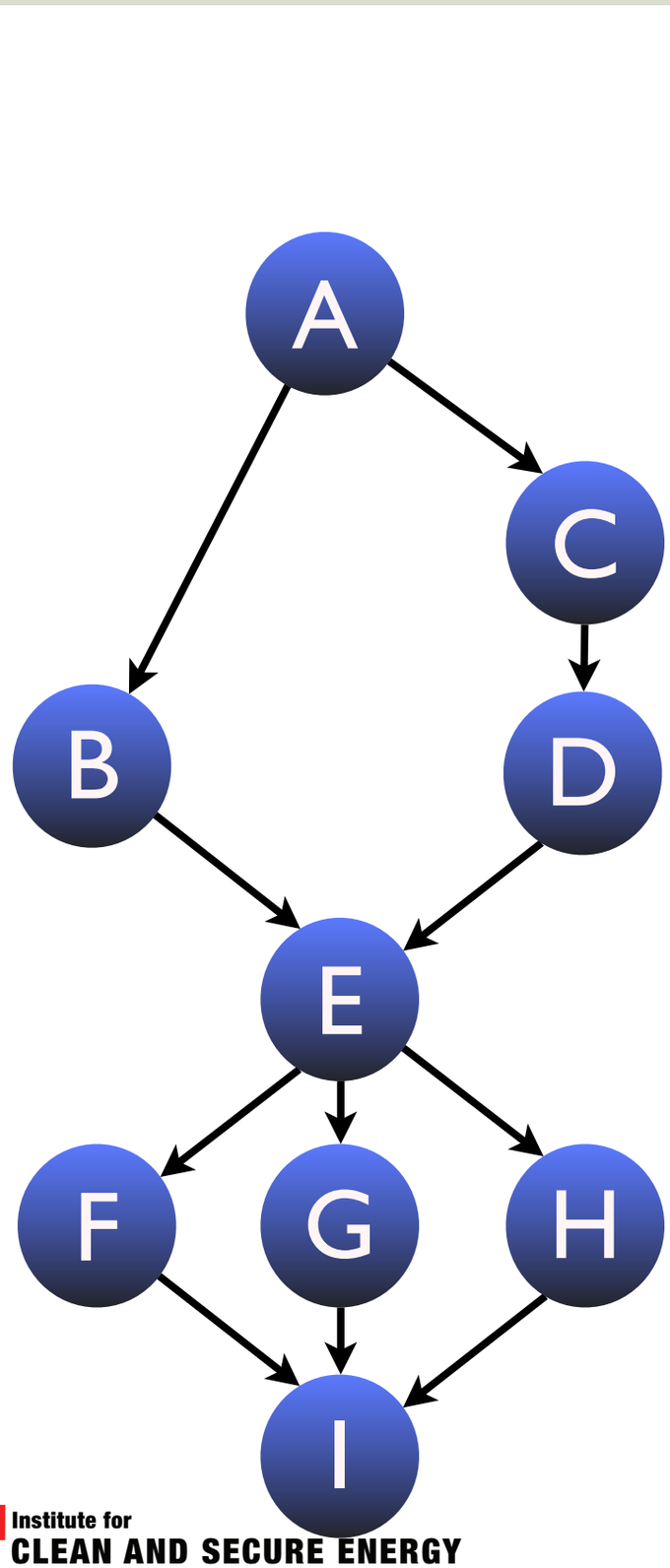


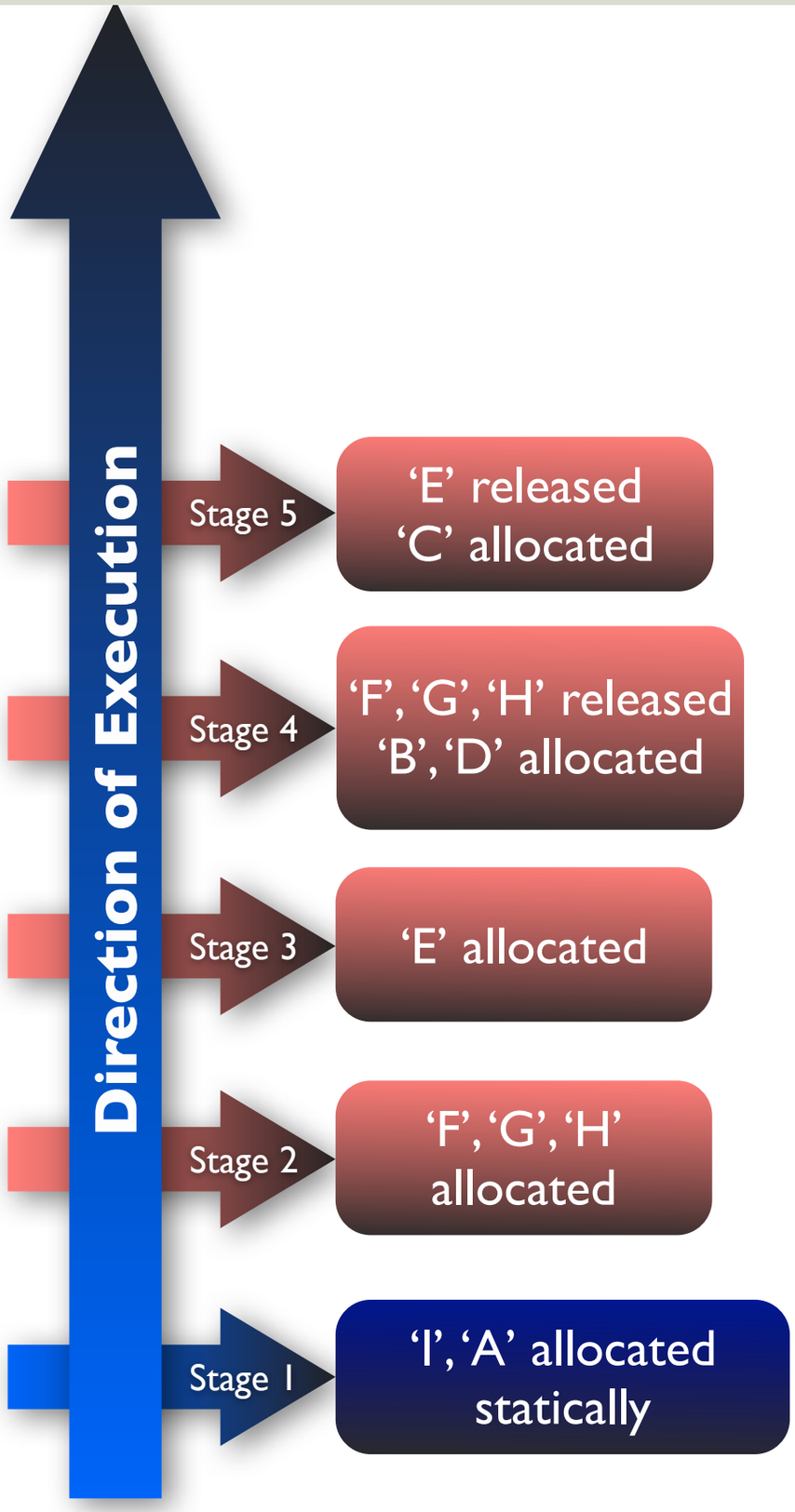


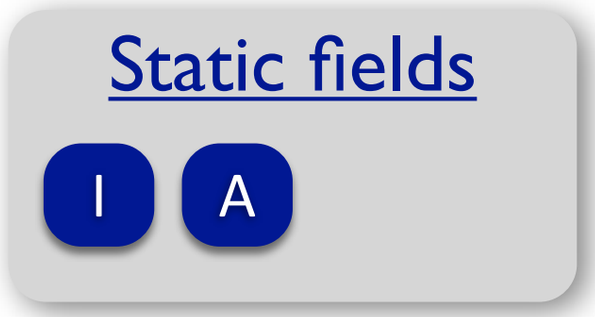


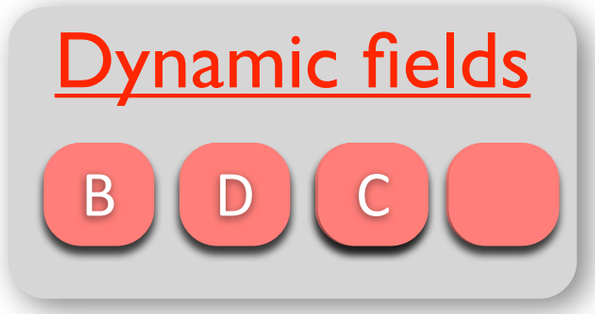


Memory Management - Pool Allocation

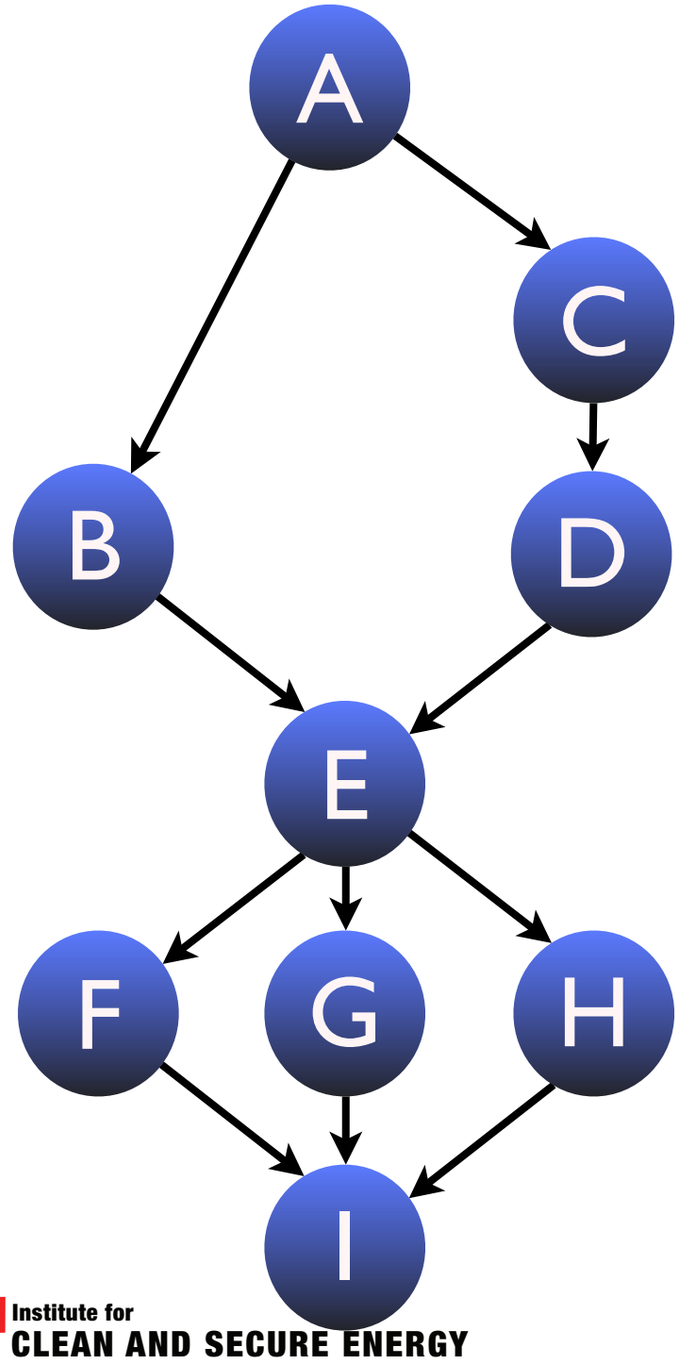


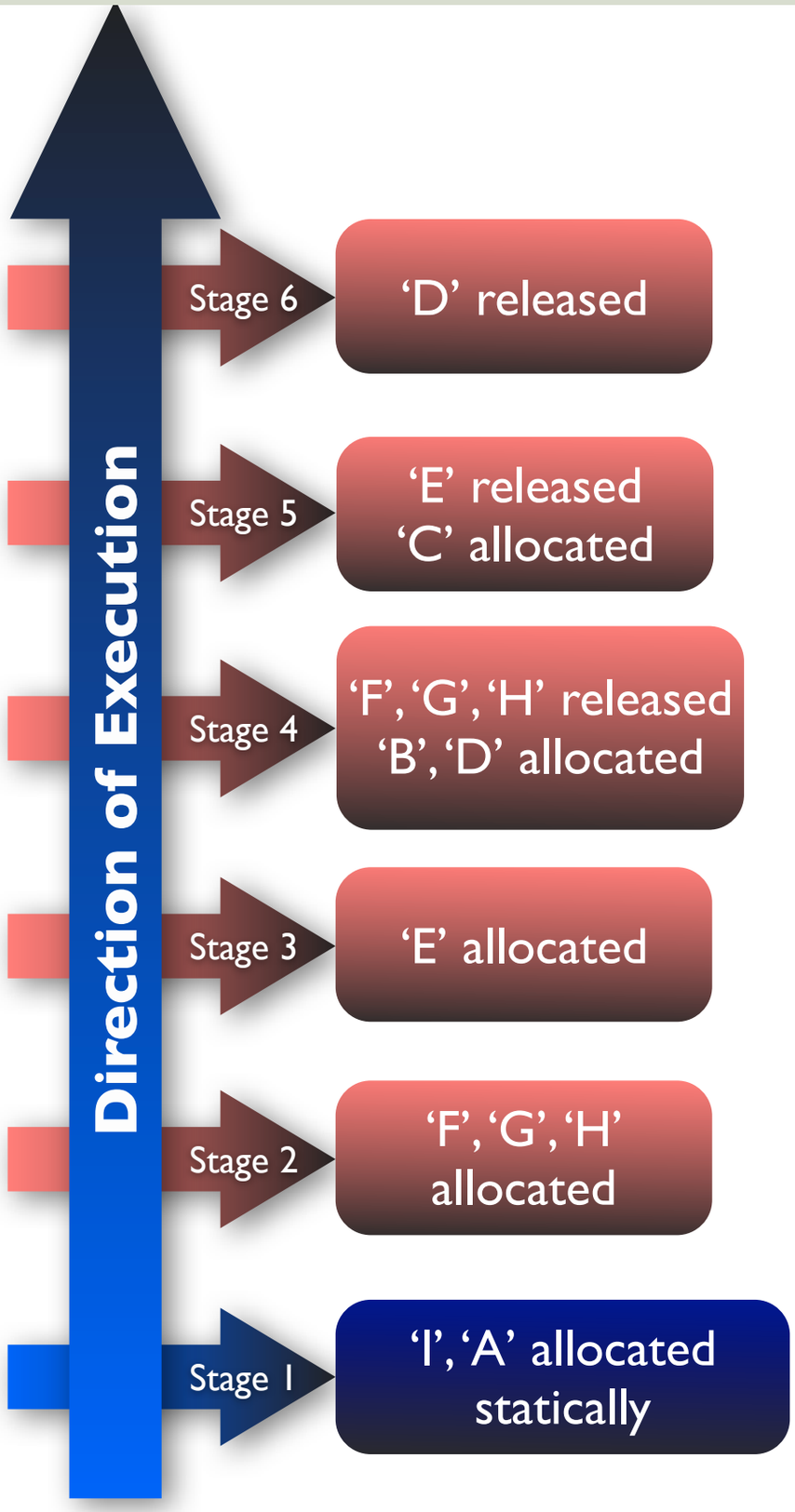


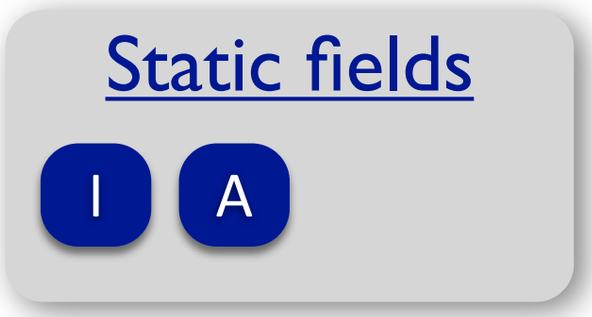


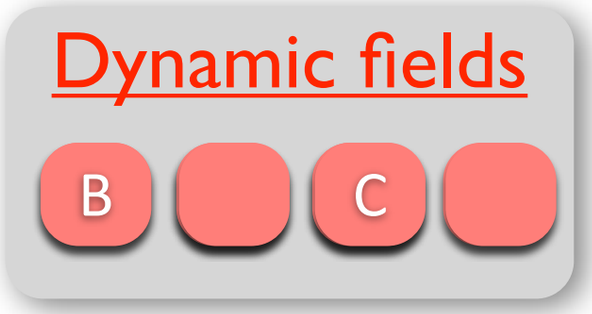


Memory Management - Pool Allocation



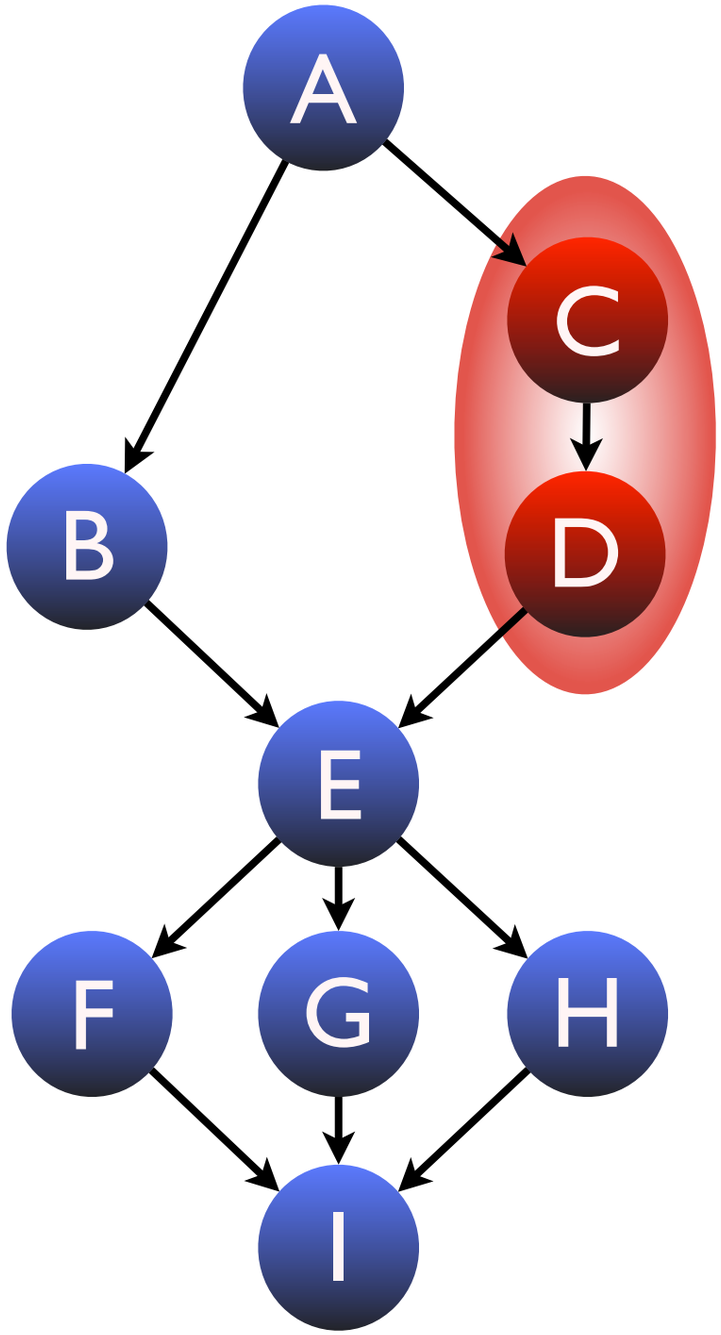






For deep graphs, this can result in very significant memory savings.

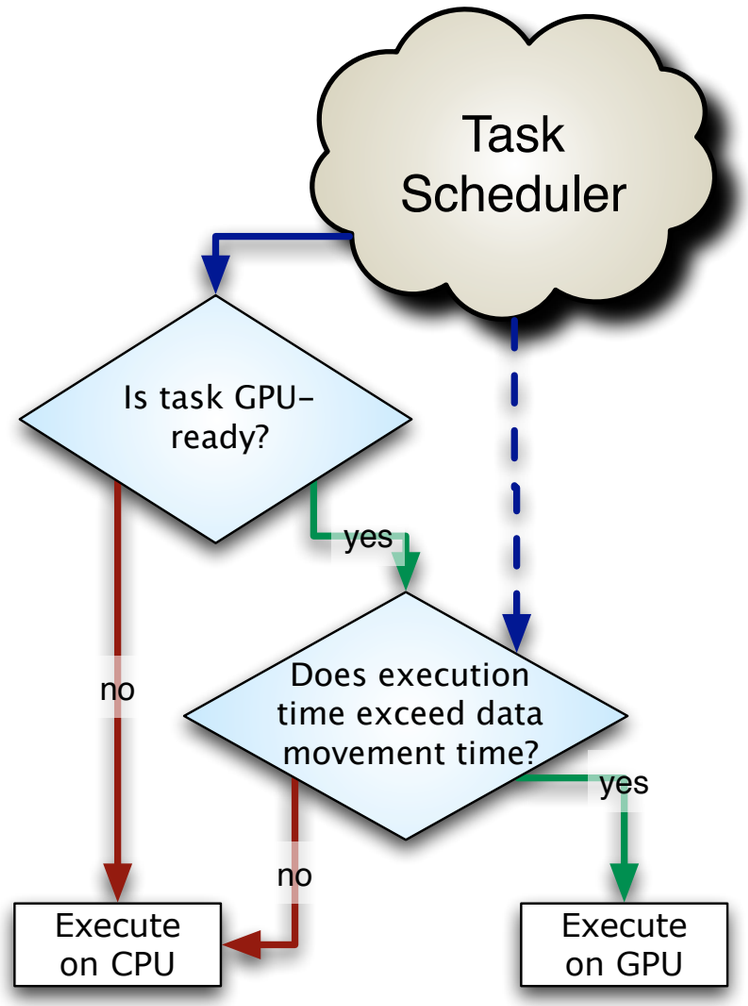
Current Work - Hybrid CPU/GPU



GPU - enabled nodes

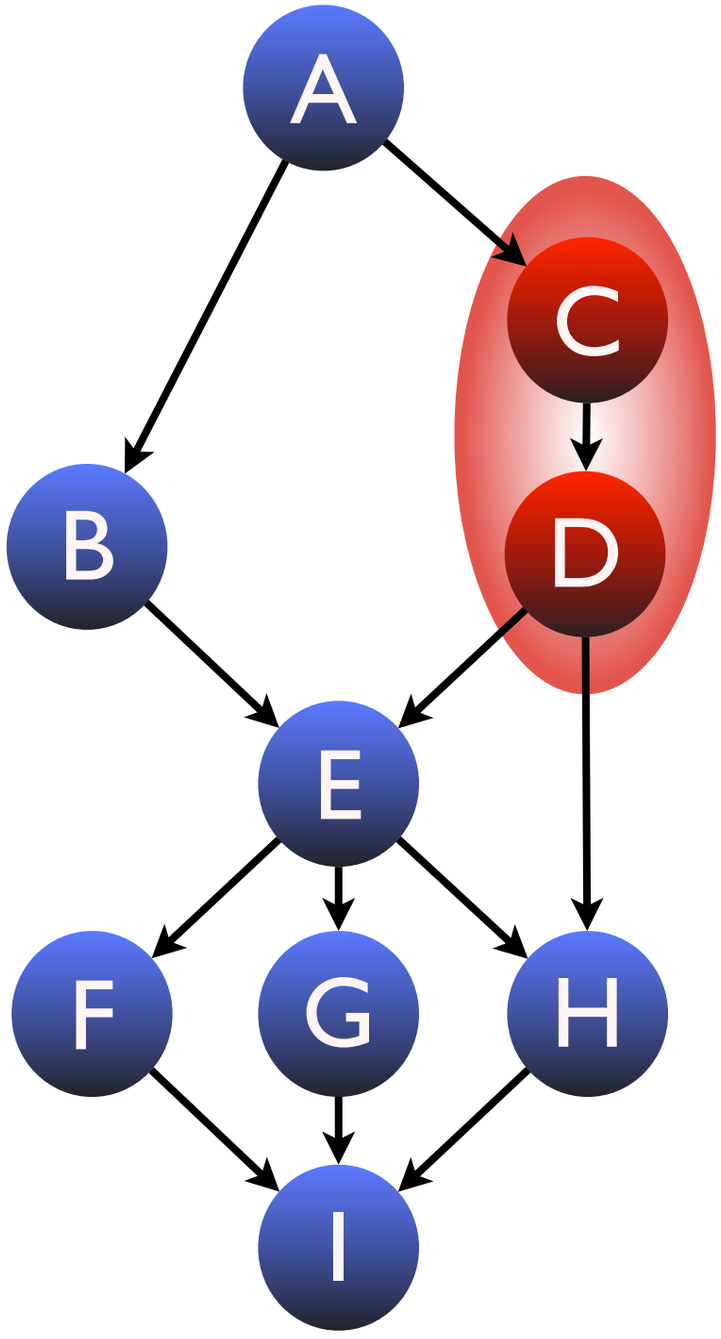
'C' and 'D' may be computed on GPU with a single copy-in/copy-out penalty.

"Graph coalescing" prevents "D" from being allocated on CPU



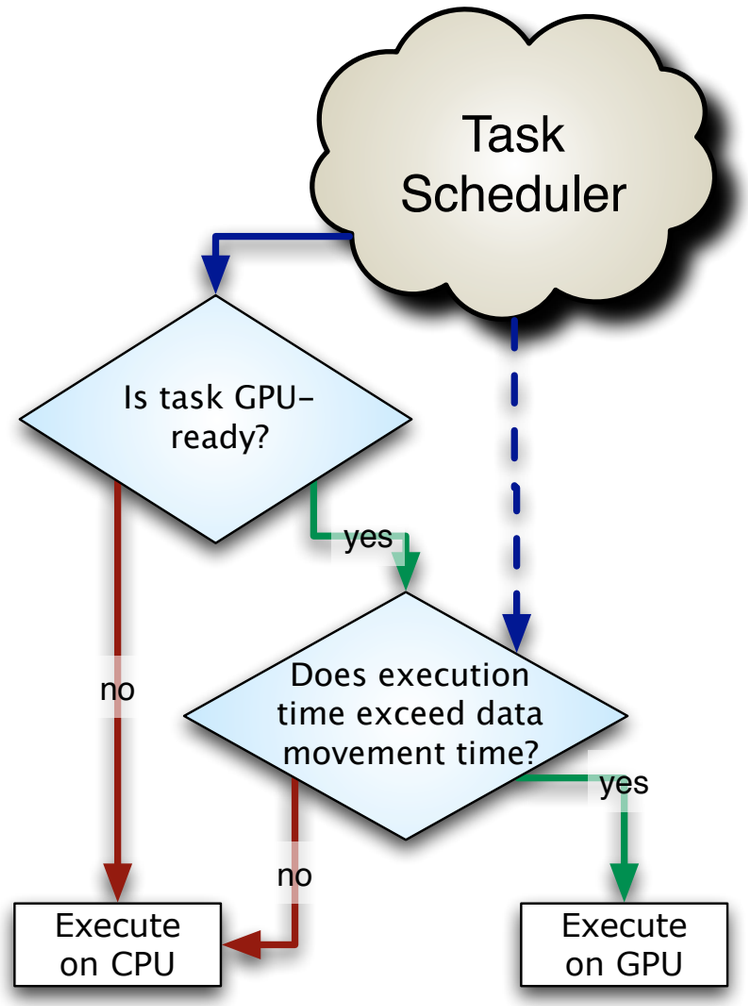
- DSL provides support for GPU deployment of operations.

Current Work - Hybrid CPU/GPU



GPU - enabled nodes

'H' may be copied while 'E' is computing, thereby hiding some latency.



- DSL provides support for GPU deployment of operations.
- Graph provides optimal scheduling of tasks across the heterogeneous system.
- Need many of tasks with lots of work (data parallel) in each.

Conclusions

- Multiple levels of parallelism will be required
- Task-based approaches can provide a great deal of insight into the structure of the problem
  - advanced memory management
  - automatic task-parallelism
  - “optimal” task scheduling in hybrid compute environments
- DSLs can simplify life (a lot!)
  - provide high-level interface to express intent, dispatch through highly optimized back-ends tuned to an architecture.
  - Programmer need not worry about architecture details, or data layout details.
  - C++ template meta programming ensures robust (correct) code.
  - Write more robust, efficient code in shorter time.
  - revise DSL back-end to affect changes through the entire code base.