Concurrent Collections: Fusion and Tiling

Chenyang Liu, Milind Kulkarni

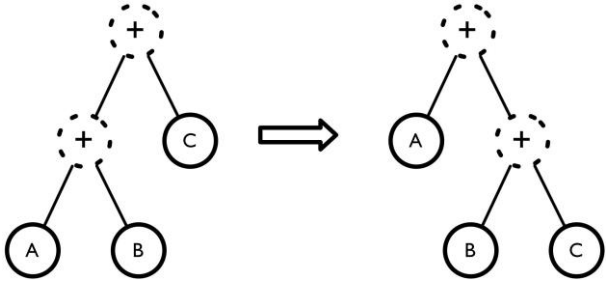
Computer Engineering

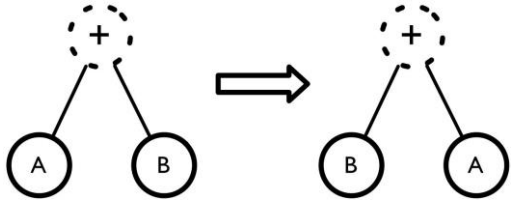
9-7-2015

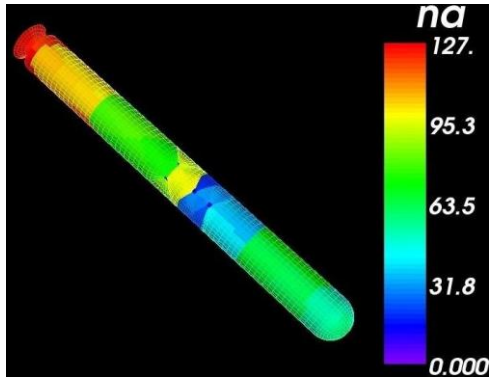
PURDUE UNIVERSITY

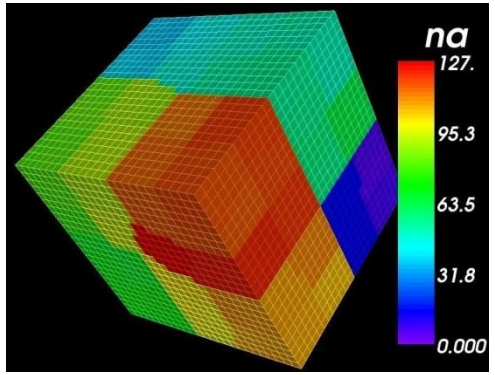
Motivation

- Previous work in recursive coupling schemes in 3D structures
  - Original problem is decomposed into smaller subdomains
  - Each subdomain is solved
  - Subdomains are coupled along interfaces
- Lessons Learned
  - Partitioning matters: Subdomain sizes and interface sizes
  - Ordering of coupling matters: commutative and associative property
  - Parallelization is difficult!









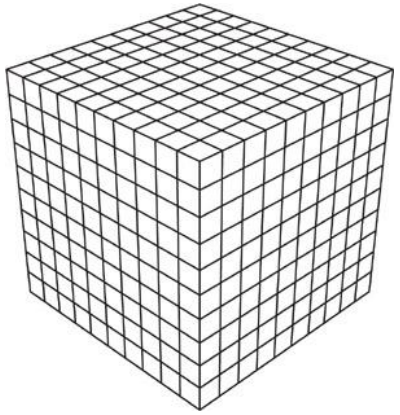
PURDUE UNIVERSITY

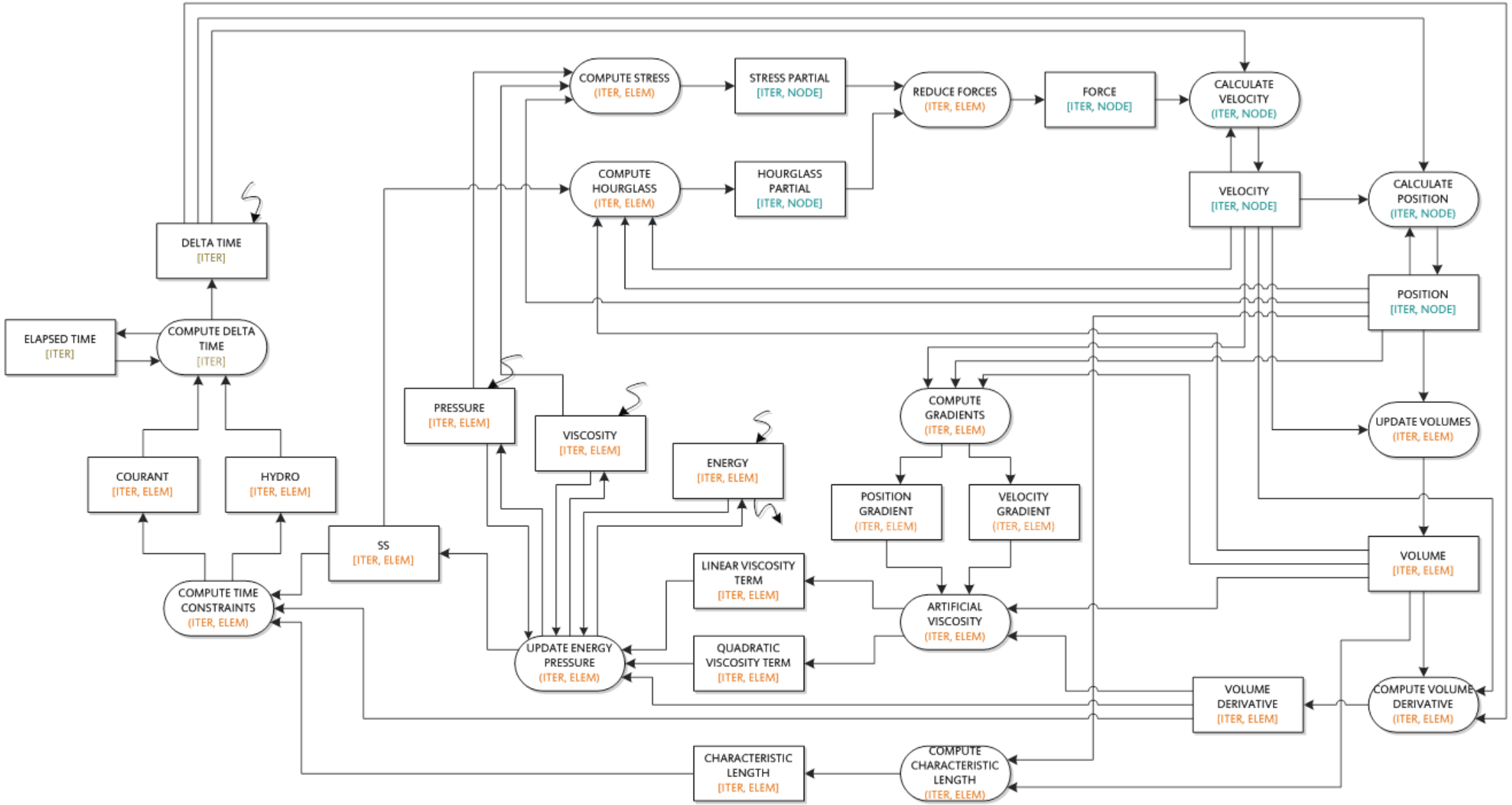
Background

- Concurrent Collections (CnC) presents a versatile framework for programming parallel applications
- Separates the concerns of domain experts and performance experts
- Express program algorithms as partially ordered computations
- Questions:
  - Are there opportunities for high-level optimizations in CnC?
  - Graph transformations
  - Fusion/Fission?
  - Tiling?

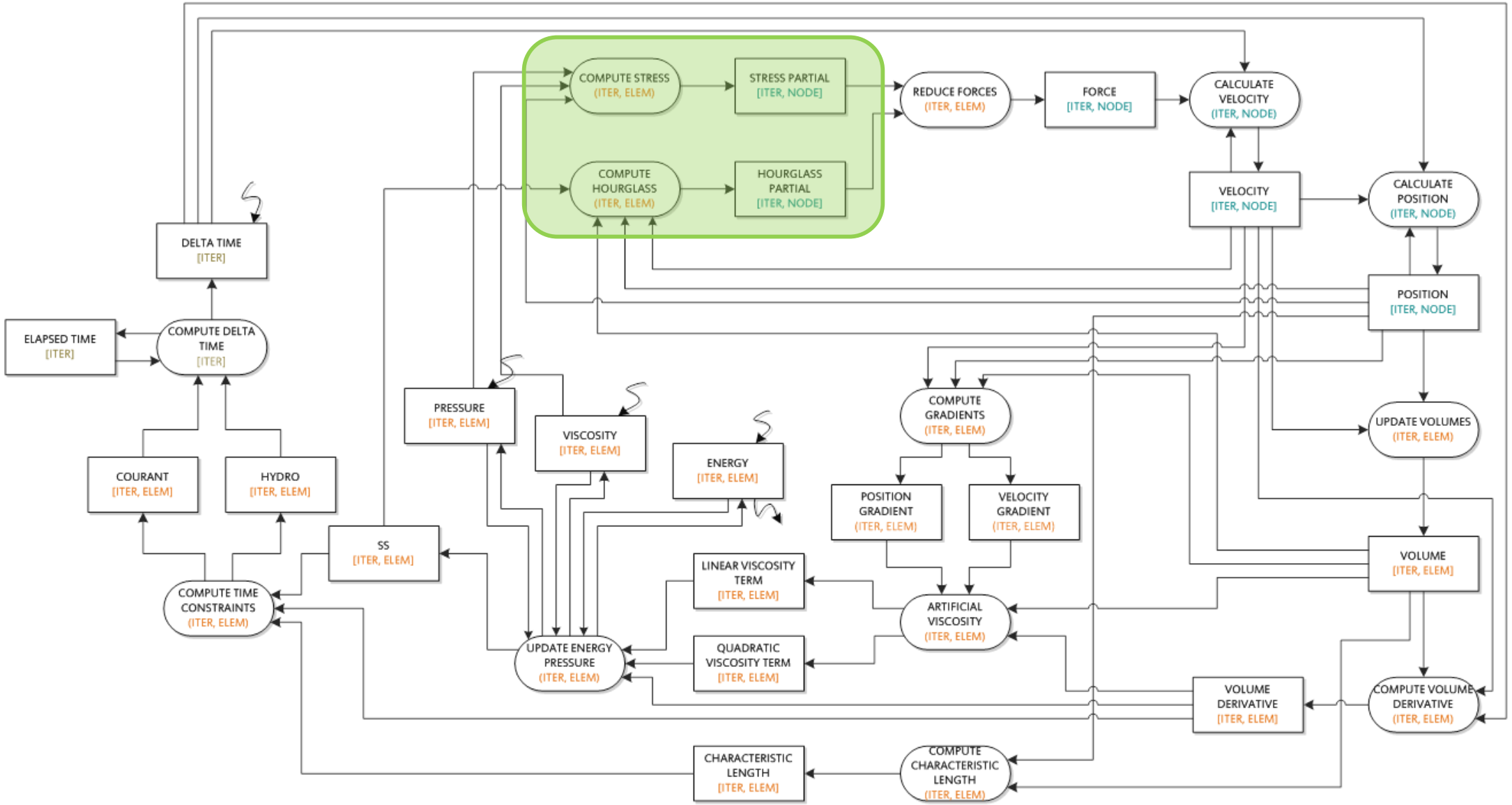
CnC LULESH

- **LULESH: Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics**
  - Challenge problem from the DARPA UHPC program
- CnC Version developed by Ellen Porter from Pacific Northwest National Laboratory (PNNL)
- 3D stencil-based program
  - Operates on a hexahedral mesh with 2 centerings:
  - Node/Element interactions/computations
  - Complex Algorithm
  - Ample Parallelism

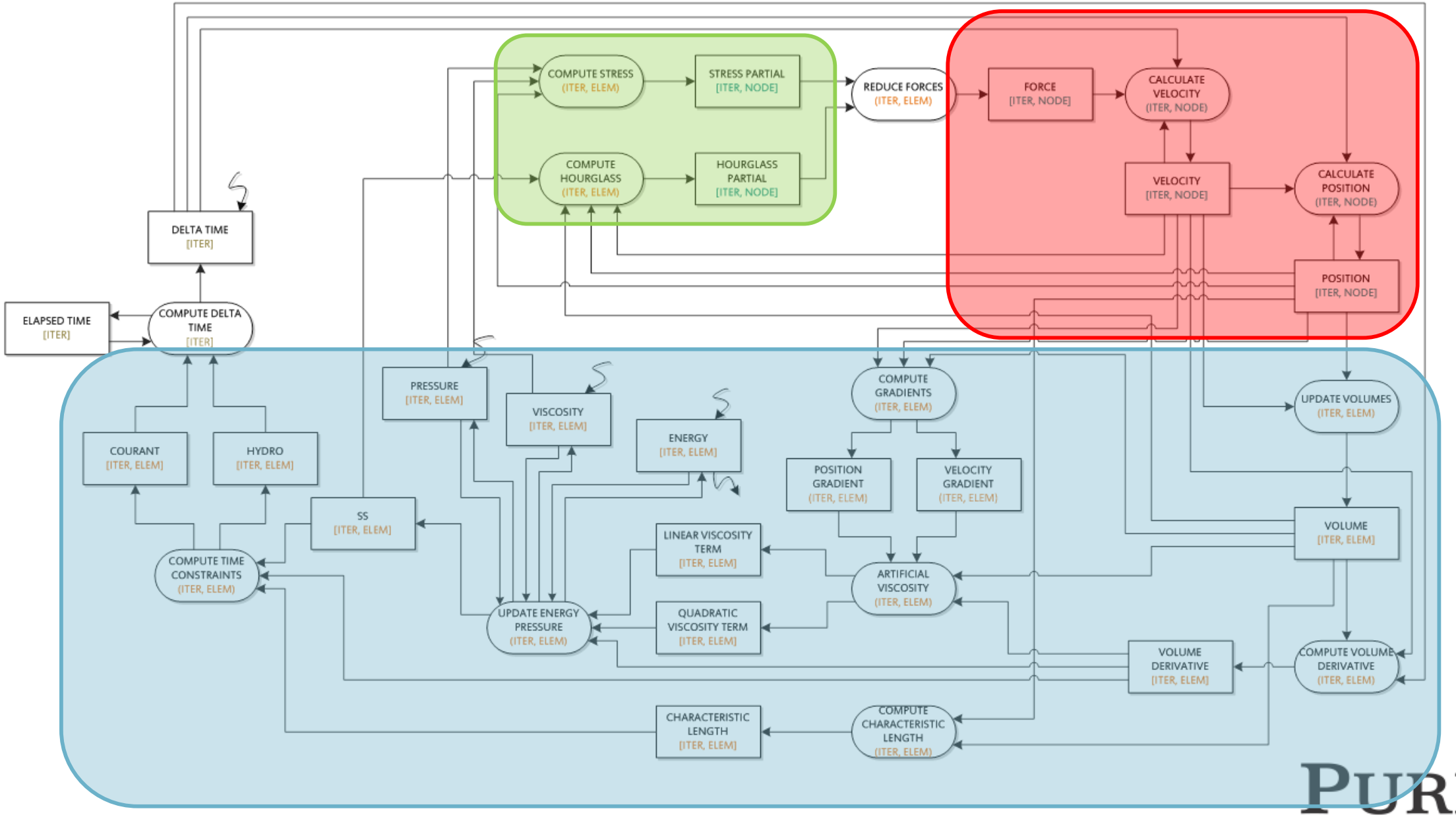




PURDUE UNIVERSITY



PURDUE UNIVERSITY



PURDUE UNIVERSITY

Problem

- Fully decomposed algorithm results in fine-grained parallelism
  - Not enough work in each step! Scheduling overheads dominate.
- Need to coarsen the computation
  - Fusion: Combine multiple steps (graph level)
  - Tiling: Combine multiple step instances from multiple tags
- Challenge:
  - When is it legal to fuse or tile?
  - How to do it?

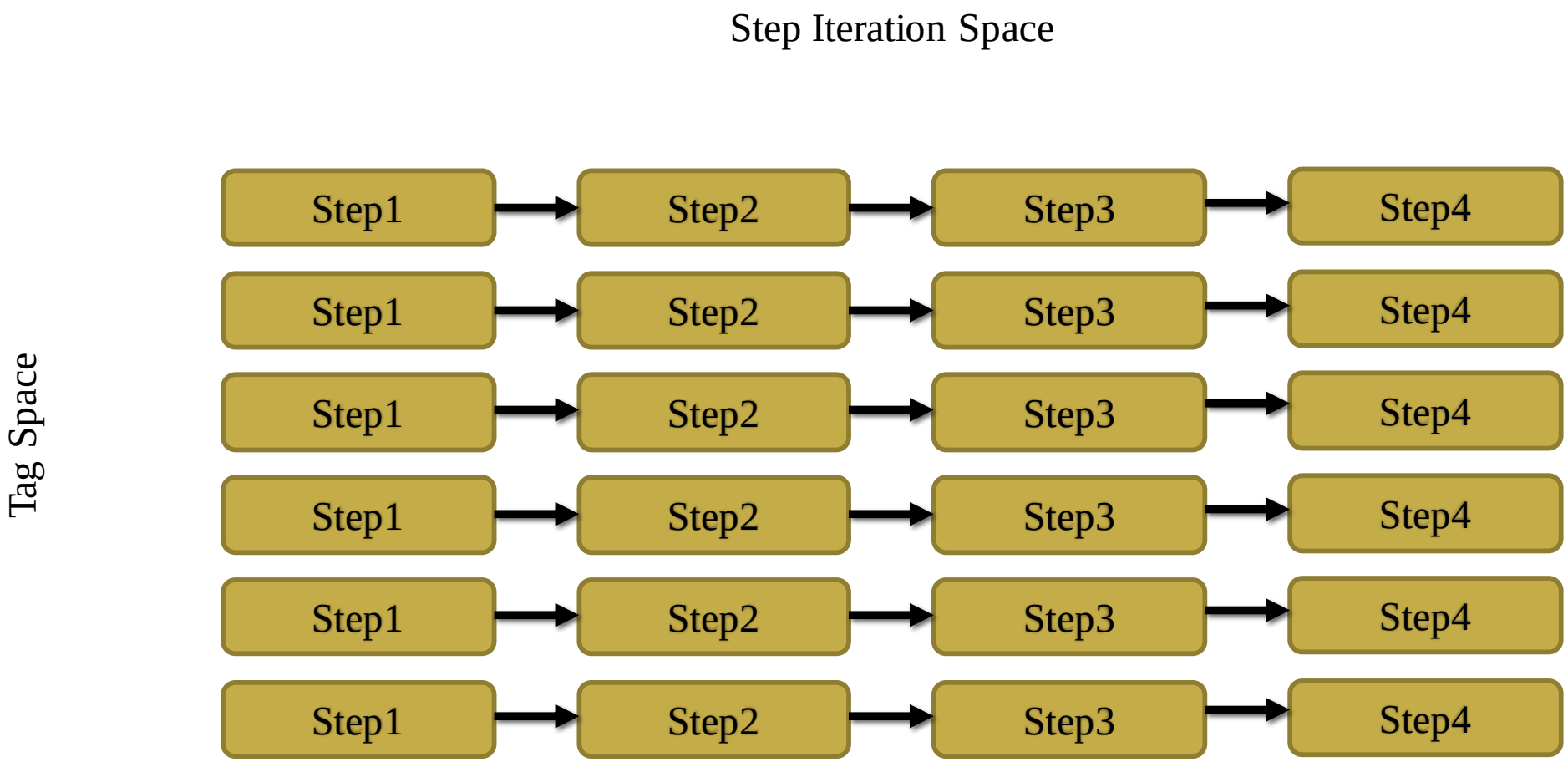
Fusion and Tiling

- When can you fuse?
  - Step collections are prescribed by the same tag, all dependencies are within each tag space
  - Dependencies become serialized. Cannot fuse if a “get” depends on value “put”d from a different tag

```
Step1: cnc.dataout.put(produced_data, my_index)
Step2: For(All neighbors) {cnc.dataout.get(consumed_data, neighbor_index)}
```

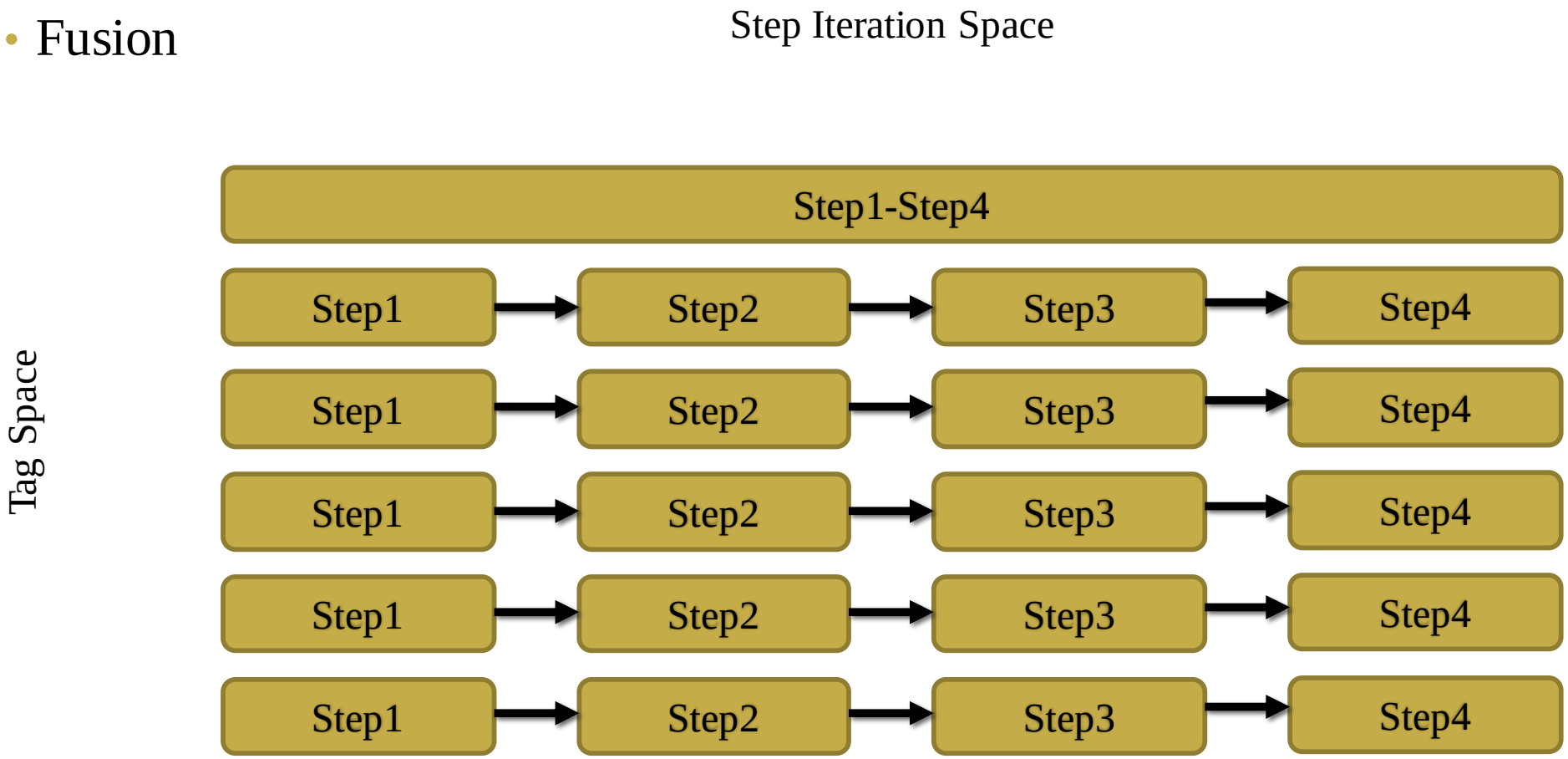
- When can you tile?
  - Step collection operates on multiple tags, performing the same work, independently (stencils)
  - Computation is serialized

Fusion/Tiling



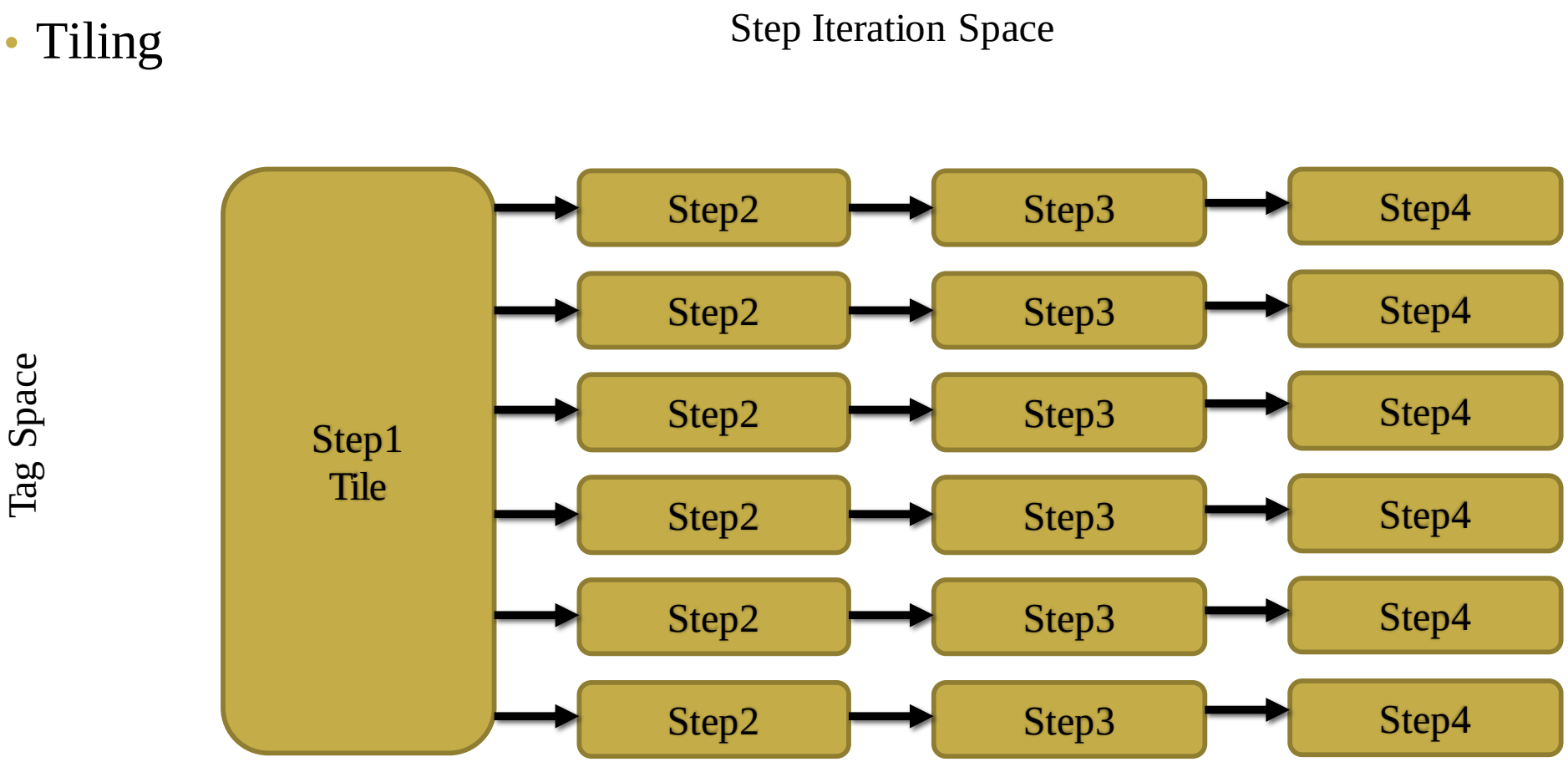
Fusion/Tiling

- Fusion

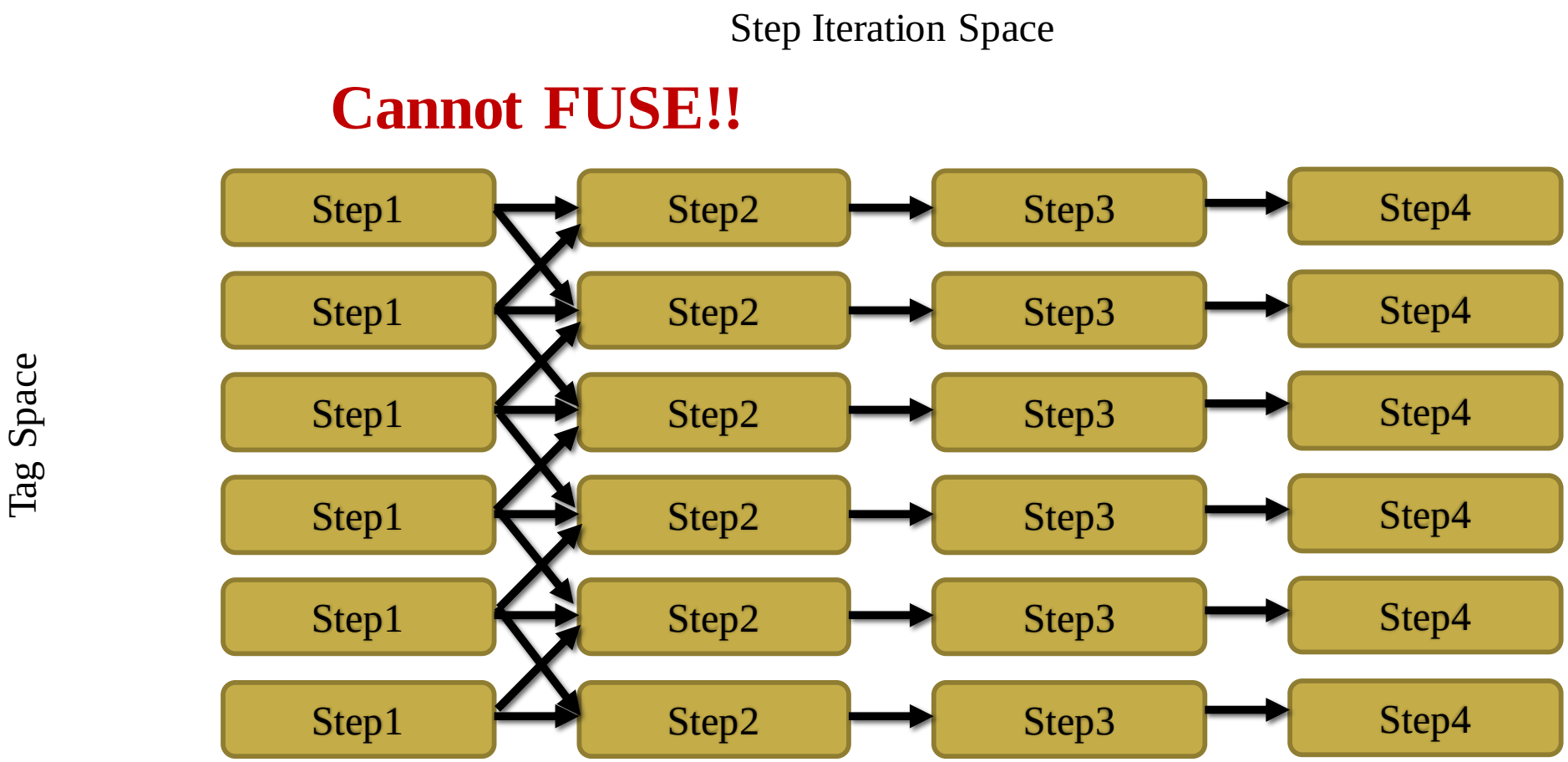


Fusion/Tiling

- Tiling



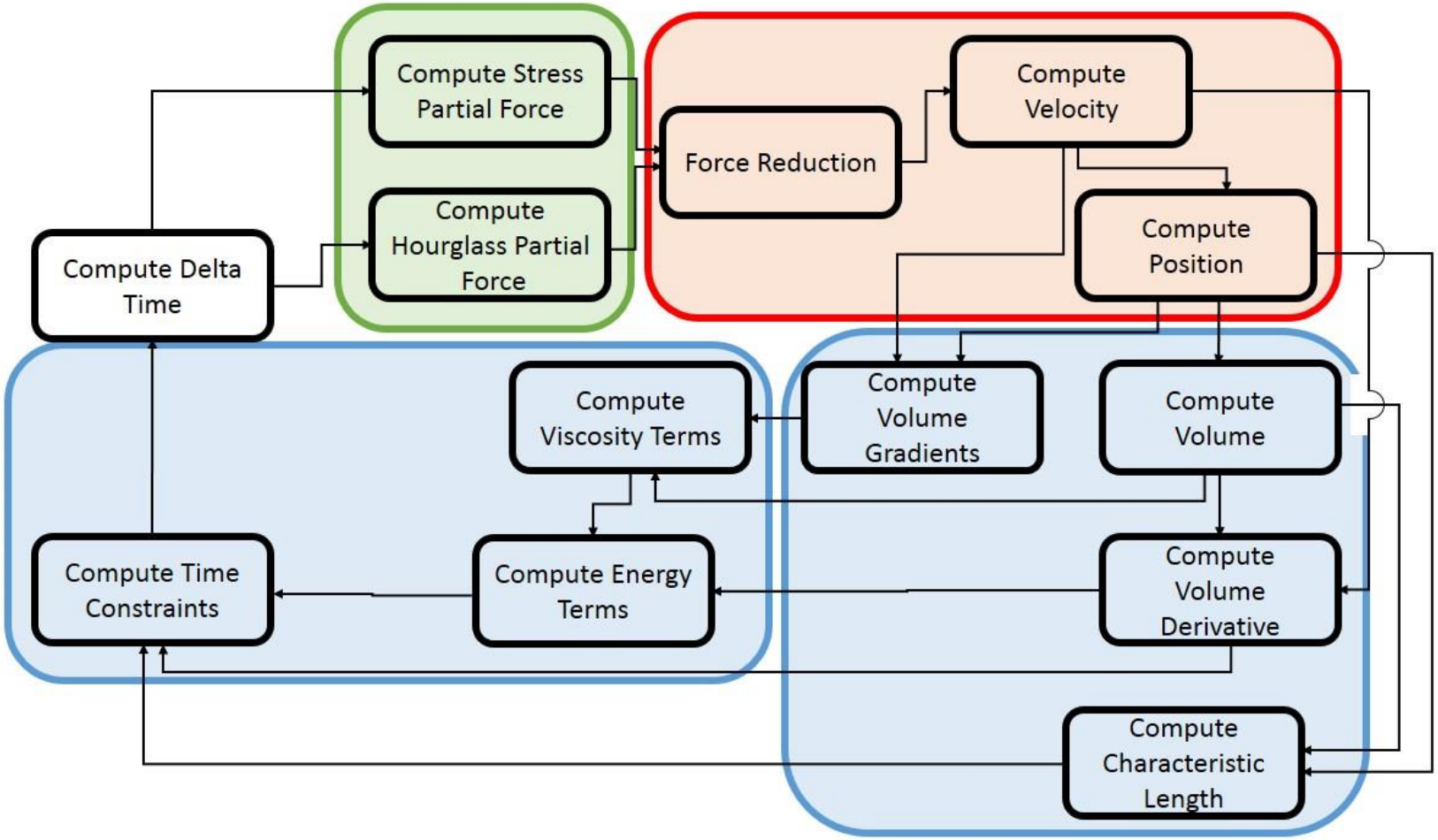
Fusion/Tiling:



Fused/Tiled Steps

- Step collections get altered
  - Usually more dependencies
    - Larger working set, temporary data
  - More computation
    - Need to allocate data/inputs efficiently
- Still need to maintain step-like behavior ('get's first, puts later)
- Tile size tuning
- Other Optimizations
  - Shared dependencies
    - Tiling may result in data reuse, especially for neighbor cases
  - Data Tiling: reduce total number of dependencies for tiled data structures (not tested)

LULESH: Fuse Algorithm

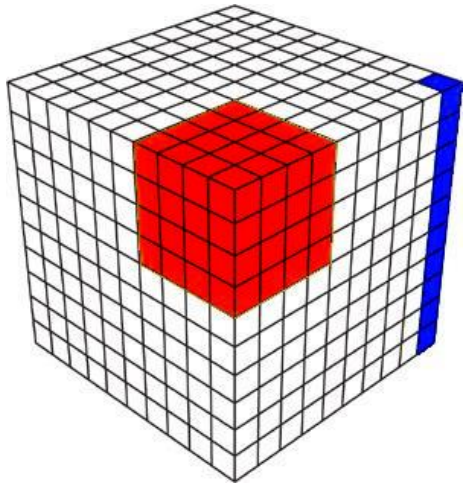


```
int reduce_force::execute(const pair & t, lulesh_context & c) const {
  int iteration = t.first;
  int node_id = t.second;
  int local_element_id;
  double delta_time;
  vector force, previous_velocity, velocity;
  vector stress_partial[8], hourglass_partial[8];
  vertex previous_position, new_position;

  c.delta_time.get(iteration, delta_time);
  c.velocity.get(pair(iteration - 1, node_id), previous_velocity);
  c.position.get(pair(iteration - 1, node_id), previous_position);

  for(local_element_id = 0; local_element_id < 8; local_element_id++) {
    int element_id = c.mesh.nodes_element_neighbors[node_id][local_element_id];
    if(element_id < 0) // If we are on a boundary
      continue;
    int map_id = (node_id << 3) | local_element_id;
    c.stress_partial.get(pair(iteration, map_id), stress_partial[local_element_id]);
    c.hourglass_partial.get(pair(iteration, map_id), hourglass_partial[local_element_id]);
  }

  force = reduce_force_work(&c, iteration, node_id, stress_partial, hourglass_partial);
  velocity = compute_velocity_work(&c, iteration, node_id, delta_time, force, previous_velocity);
  new_position = compute_position_work(&c, iteration, node_id, delta_time, velocity, previous_position);
  return CnC::CNC_Success;
}
```

PURDUE UNIVERSITY

Experimental Results

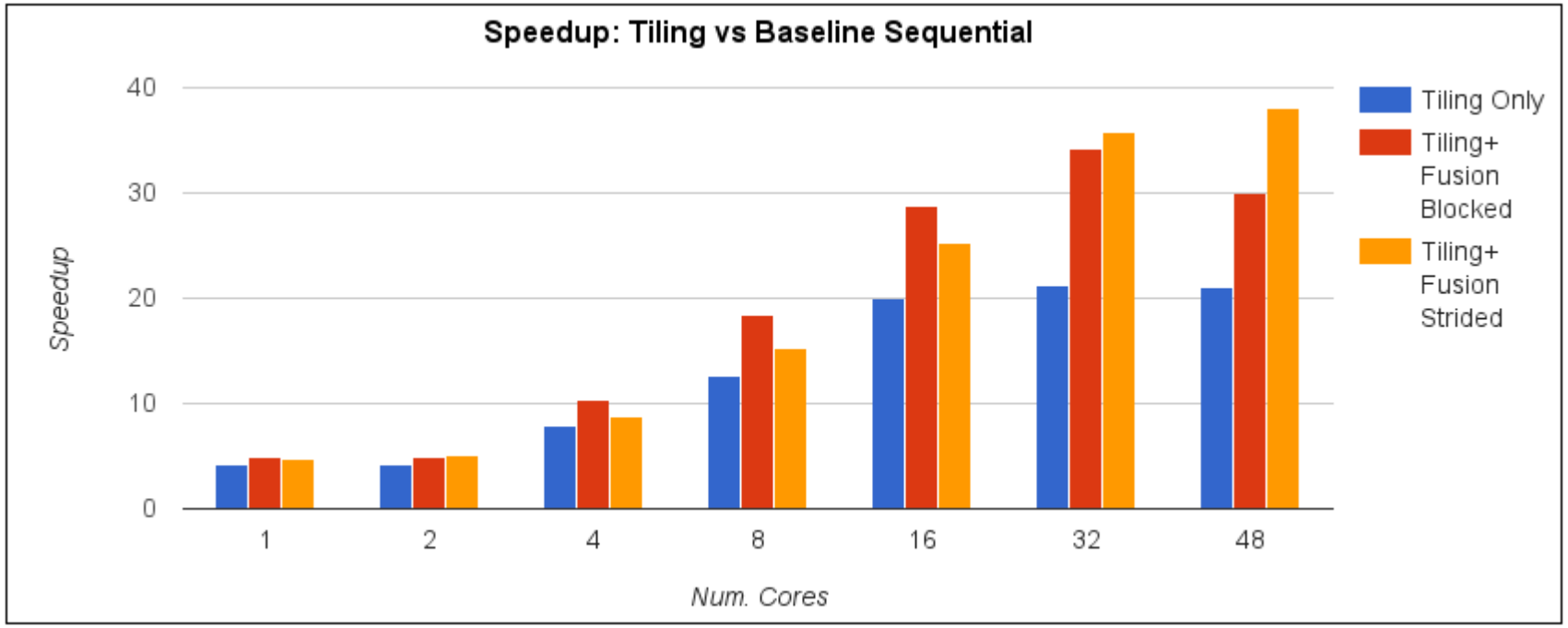
- AMD Opteron 6176 SE system with four 12-core processors (48 cores total) running at 2.3 GHz.
- Experiments
  - Baseline
  - Fused-only
  - Tiled-only
  - Fused+Tiled Blocked (red)
  - Fused+Tiled Strided (blue)



Experimental Results cont.

Table 1: Timing Results: 60^3 Sized Mesh

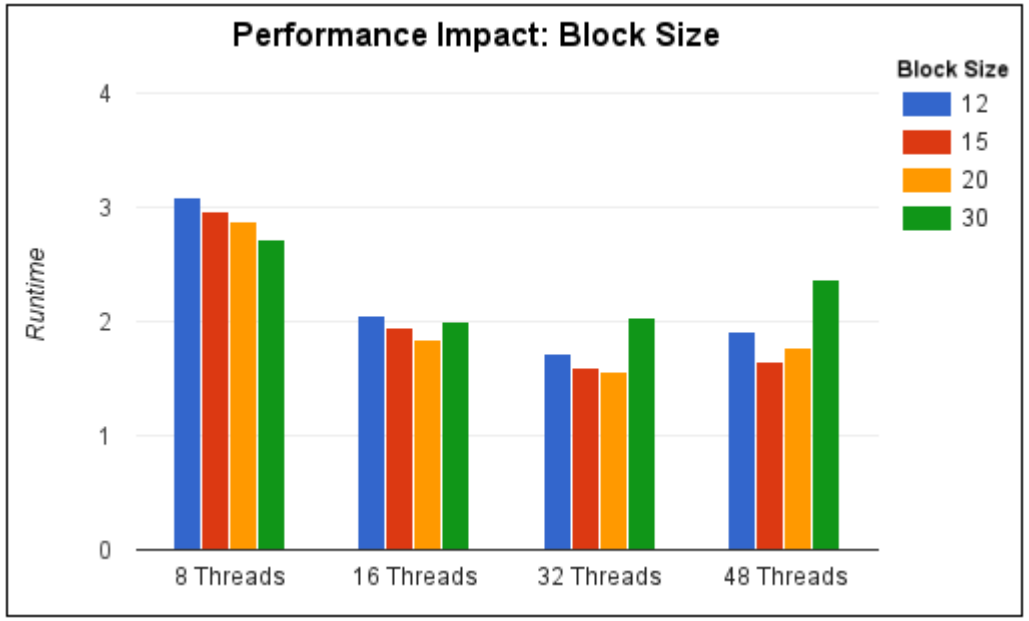
	Number of Cores						
	1	**2**	**4**	**8**	**12**	**32**	**48**
Baseline	53.180	52.900	71.852	108.531	110.515	110.572	119.466
Fusion Only	26.288	26.405	33.3389	52.224	51.391	54.340	57.430
Tiling Only	12.67596	12.5693	6.677482	4.234	2.652	2.504	2.526
Blocked Fuse-Tile	10.749	10.699	5.110	2.883	1.845	1.557	1.768
Strided Fuse-Tile	11.171	10.610	6.025	3.498	2.104	1.483	1.397



PURDUE UNIVERSITY

Tiling: Block Size

- Parameter: block size
  - Smaller size creates excess fine-grain parallelism
  - Larger size limits available parallelism
  - Sweet spot

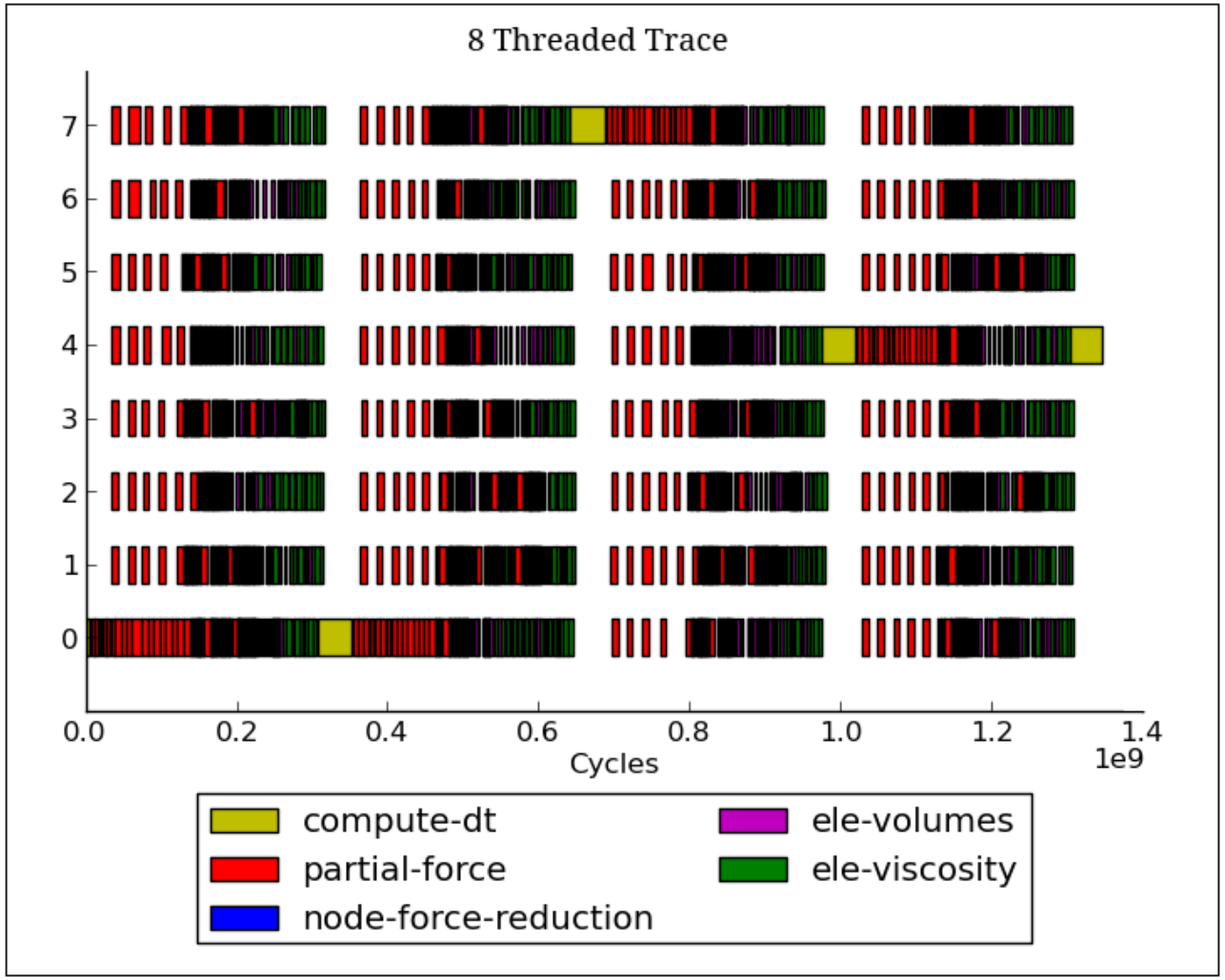


Future Goals

- Automatic transformations for tiling/fusion
  - CnC spec (graph) level data is insufficient for determining transformation legality
- Data Tiling optimizations
- Other scientific applications
- Hierarchical CnC

Thanks!

Runtime Trace



PURDUE UNIVERSITY