Correctness of Dynamic Dependence Analysis for Implicitly Parallel Tasking Systems

Wonchan Lee
Stanford University

George Stelle
LANL

Patrick McCormick
LANL

Alex Aiken
Stanford University

November 12, 2018 @ Correctness'18

Implicitly Parallel Tasking Systems

- Automatically extract parallelism from program using dynamic dependence analysis

Program

```
task F(x)
task G(x)

while (*):
  F(A[0])
  F(A[1])
  if (*):
    G(A[h(0)])
    G(A[h(1)])
```

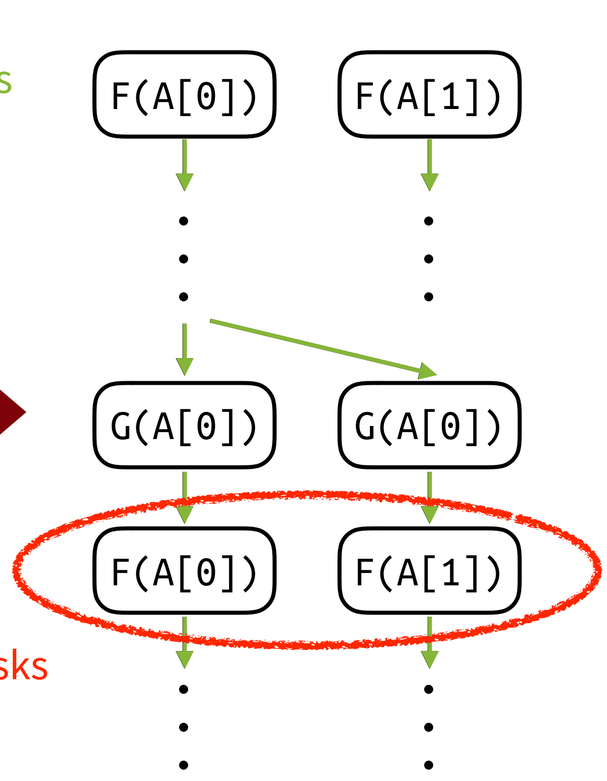
Exec.

Task stream

F(A[0]) F(A[1]) • • • G(A[0]) G(A[0]) • • •

Dependence Analysis

Task graph



- Legion, PaRSEC, StarPU, PyTorch, etc.

Benefits of Implicitly Parallel Tasking Systems

- Sequential semantics
  - No parallel programming errors
- Precise task dependencies in the presence of dynamic behaviors

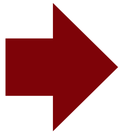
```
task F(x)
task G(x)

while (*):
  F(A[0])
  F(A[1])
  if (*):
    G(A[h(0)])
    G(A[h(1)])
```

control divergence

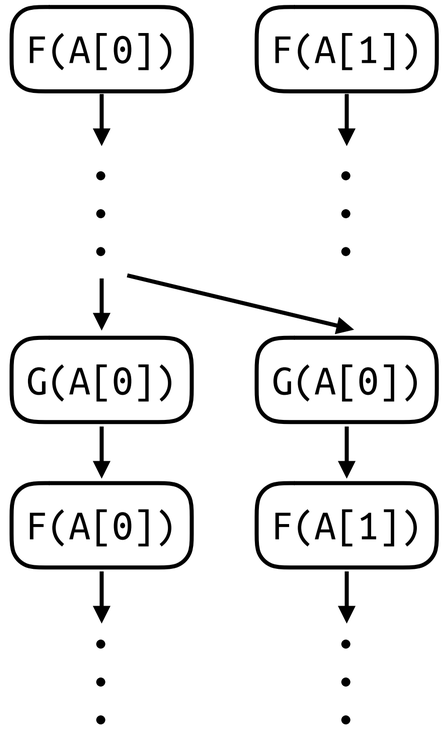
opaque function calls

F(A[0]) F(A[1]) • • • G(A[0]) G(A[0]) • • •

no conditionals or function calls



captures dependencies precisely



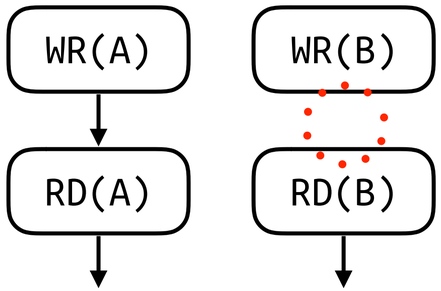
Correctness of Dynamic Dependence Analysis

- Important
- Non-trivial to achieve
  - Due to optimizations in real algorithms
- Topic of this paper
  - D_{epoch}: a model dependence analysis algorithm using *epochs*
  - Correctness proof for D_{epoch}

Soundness and Completeness

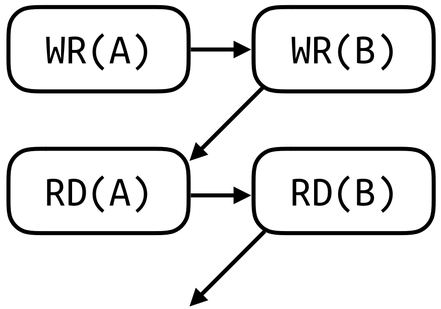
Task stream: `WR(A) WR(B) RD(A) RD(B)`

- Soundness
  - All dependencies are discovered



data race on B
→ unsound 😕

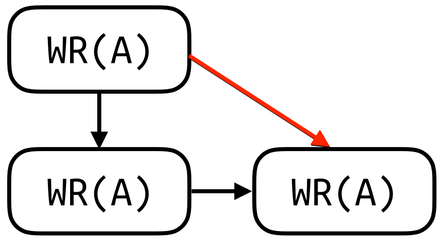
- Completeness
  - No independent tasks are serialized unnecessarily



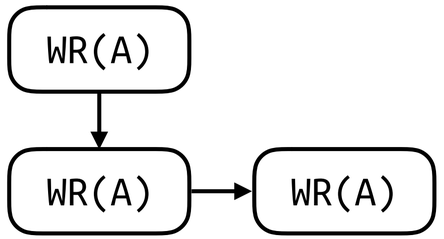
serial execution
→ incomplete 😅

Bonus Point: Parsimony

- Parsimonious task graph
  - One with fewest edges among equivalent task graphs



has transitive edge
→ not parsimonious 😐

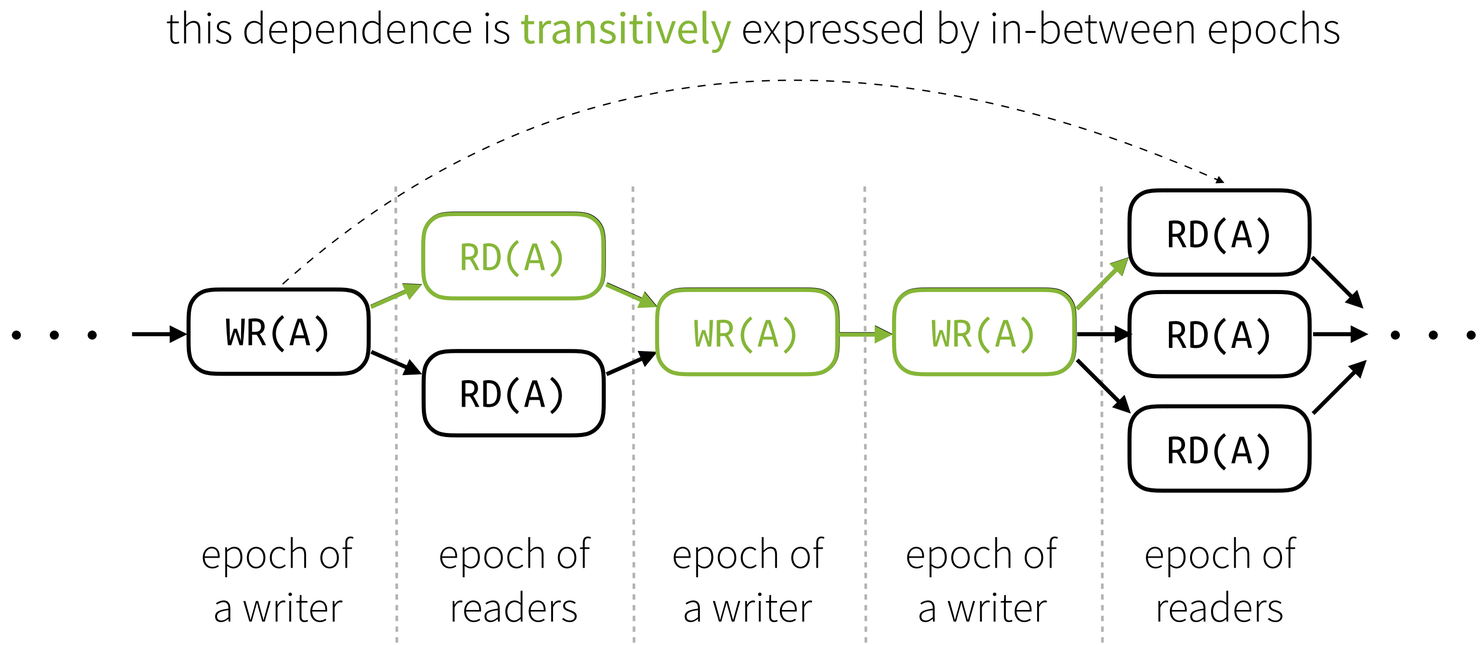


no transitive edge
→ parsimonious 👍

- Task graphs are often not parsimonious, because they require costly transitive reduction
  - Still valuable to understand when algorithms produce parsimonious task graphs

D_{epoch}: Epoch-based Algorithm

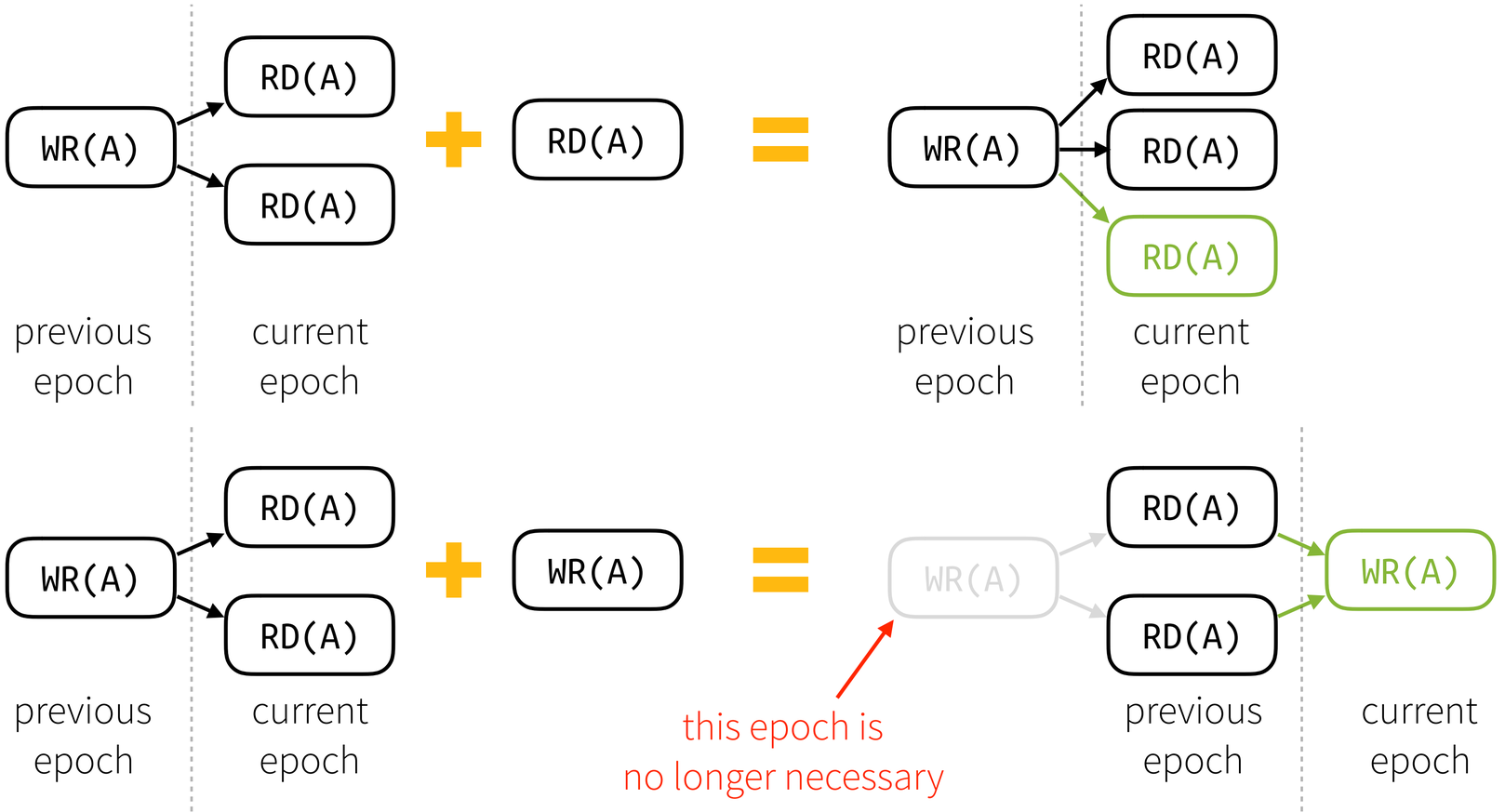
- Based on SWMR invariant: either **S**ingle **W**riter or **M**ultiple **R**eaders can exist for a region
  - Tasks accessing a single region form *epochs* of independent tasks



- Only the dependencies between immediate epochs are non-transitive

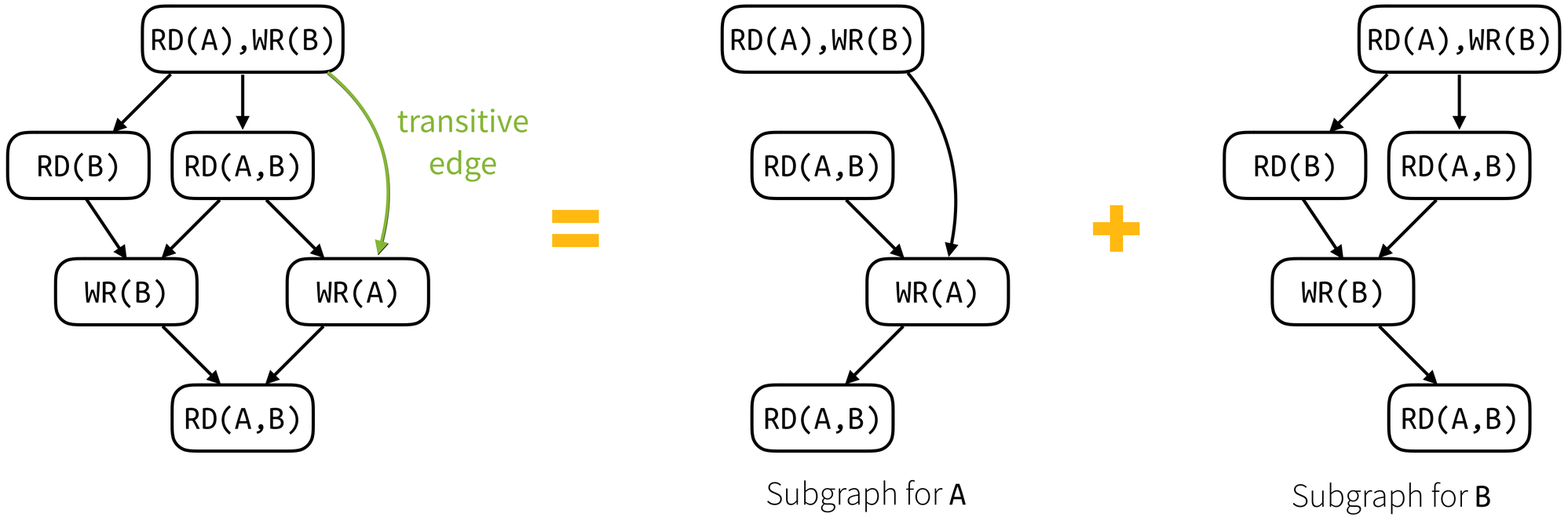
D_{epoch}: Epoch-based Algorithm

- Based on SWMR invariant: either **S**ingle **W**riter or **M**ultiple **R**eaders can exist for a region
- "Two-epoch rule": remembering only the last two epochs is enough to find all dependencies



D_{epoch}: Epoch-based Algorithm

- Key idea: a task graph is decomposed into subgraphs for individual regions



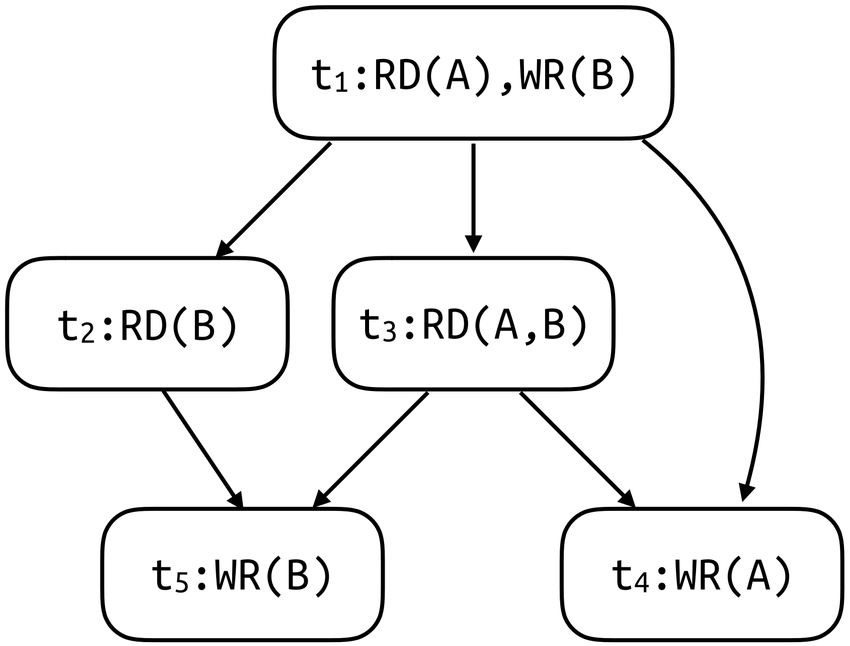
- Track epochs individually

Subgraphs have no transitive dependencies

  - Each subgraph do not have transitive dependencies, but the union might

Example

Task graph



Task

t_1: RD(A),WR(B)

t_2: RD(B)

t_3: RD(A,B)

t_4: WR(A)

t_5: WR(B)

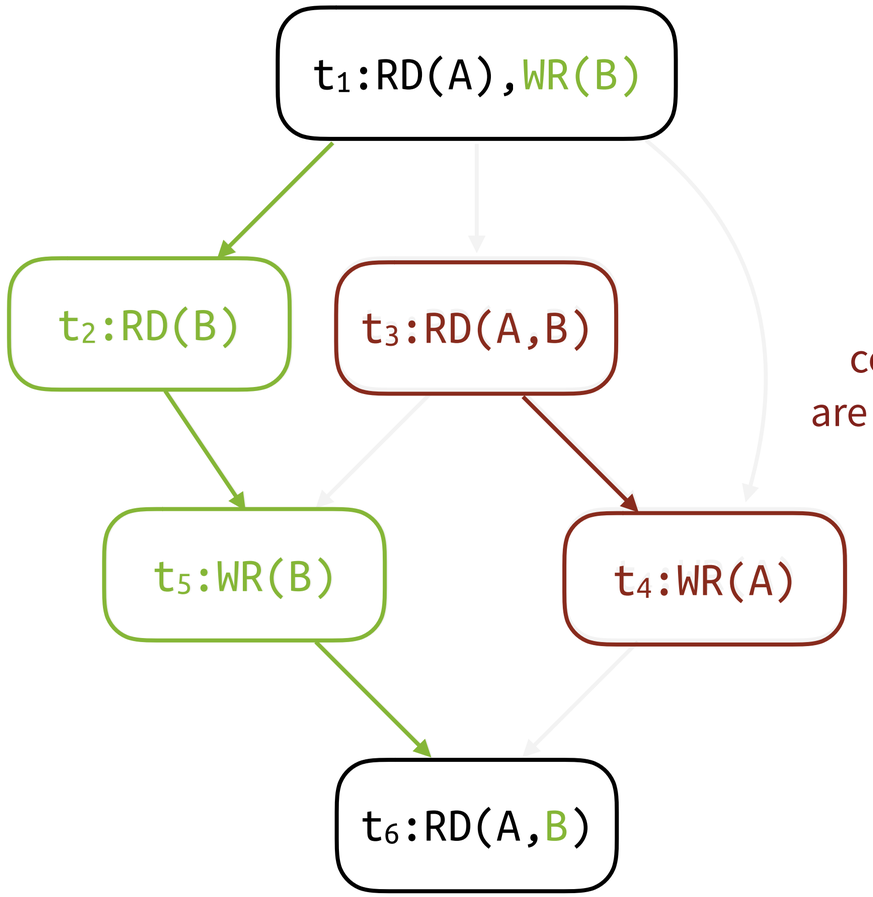
Region A		Region B	
previous epoch	current epoch	previous epoch	current epoch
	t_1		t_1
	t_1	t_1	t_2
	t_1, t_3	t_1	t_2, t_3
t_1, t_3	t_4	t_1	t_2, t_3
t_1, t_3	t_4	t_2, t_3	t_5

Soundness and Completeness of D_{epoch}

- By showing D_{epoch} is equivalent to a naive method D_{simple}
  - $D_{simple}(t_1, \ldots, t_n)$ = connect every two tasks t_i and t_j $(i < j)$ when t_i and t_j are dependent
  - D_{simple} is sound and complete
- For every edge in D_{simple}, we show there is a path in D_{epoch} and vice versa

Soundness and Completeness of D_{epoch}

- Any two dependent tasks are connected by at least one path

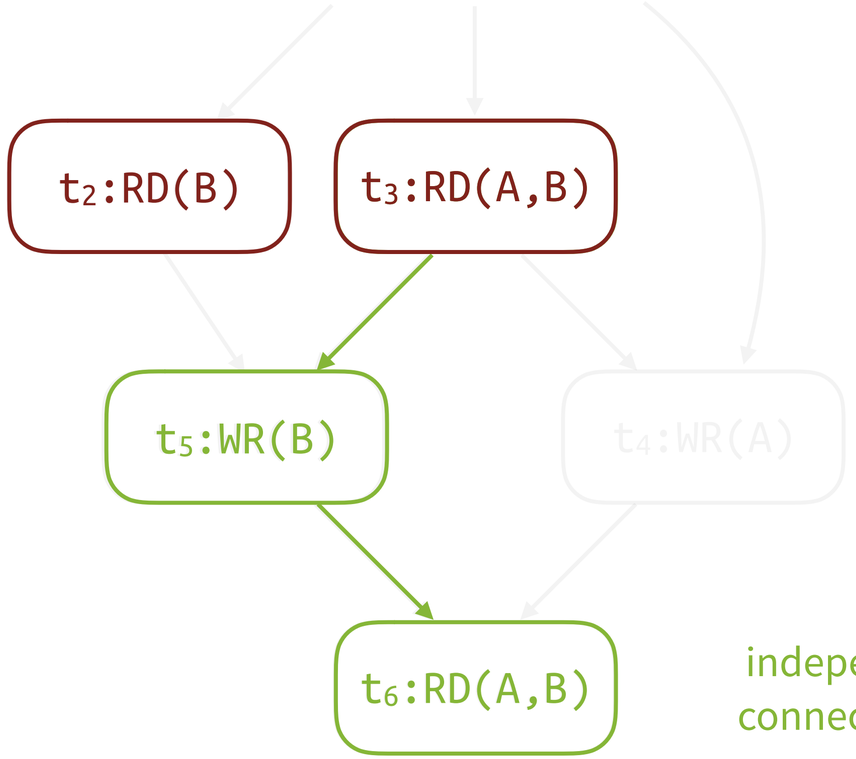


consecutive dependent tasks
are directly connected via epochs

Soundness and Completeness of D_{epoch}

- Independent tasks are never directly connected
  - The length of every path between independent tasks is always greater than 1

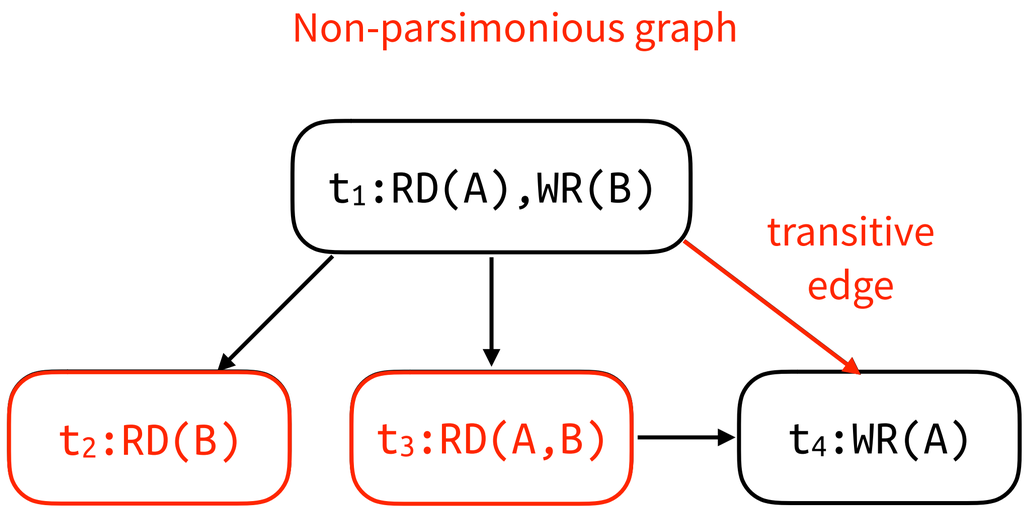
consecutive independent tasks must be mutually unreachable

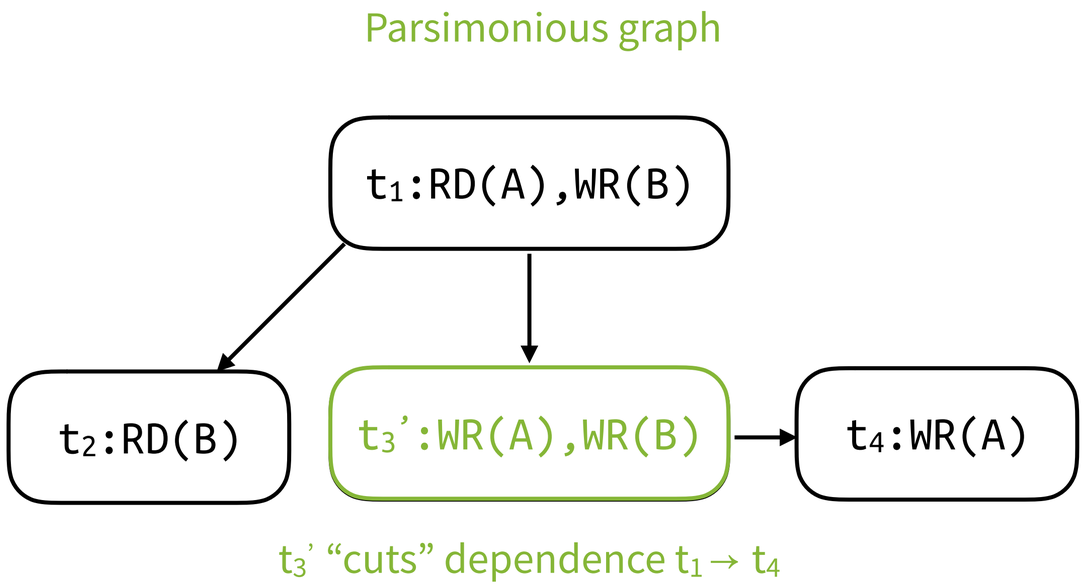


independent tasks are always connected via some other task

When D_{epoch} Produces Parsimonious Task Graphs

- When no "triangle" exists in a task stream





- For any subsequence t_1, …, t_n, if t_1 and t_n are dependent because of R, there must be a t_i that writes R

Conclusion

- Epoch-based dependence analysis algorithm D_{epoch} is sound and complete
  - And sometimes produces parsimonious task graphs
  - Even when the task graph is concurrently mutated for execution (see the paper)

Future Work

- Add advanced features in Legion to D_{epoch}
  - Coherence: multiple instances per region
  - Aliasing: regions can be aliased
  - Tracing: task graphs can be memoized
  - Resilience: failed tasks can be recovered
  - Replication: dependence analysis can be replicated

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, award DE-NA0002373-1 from the Department of Energy National Nuclear Security
Administration, NSF grant CCF-1160904, and an internship at Los Alamos National Laboratory.

Questions?