# Deadlock Avoidance in Parallel Programs with Futures

## Why Parallel Tasks Should Not Wait for Strangers

Tiago Cogumbreiro, Rishi Surendran, Francisco Martins,
Vivek Sarkar, Vasco Vasconcelos, Max Grossman

OOPSLA, Vancouver, 2017

# Futures

## Fork-join model + Data

# Widespread use of futures

## 1. Asynchrounous programming

- Language support (`async`, `await`): Python, Javascript, Rust

## 2. Task parallel programming

- Language support: Java, C#, C++, Kotlin
- Library support: C++ (TBB, Kokkos, Charm++), Java (HJ-Lib, Quasar)

# Uses of futures with shared memory

## Task-DAG parallelism

- **Data-flow** parallelism
- **Shared** collections of futures (matrices)

# Uses of futures with shared memory

## Task-DAG parallelism

- **Data-flow** parallelism
- **Shared** collections of futures (matrices)

Problem: Cyclic data-dependencies cause *deadlocks*!

Off-by-one errors cause deadlocks

# Shared memory and futures

**Intuition:**

> Root-cause of future-deadlocks are data races.

1. Is it true?
2. Why?
3. How can we use this property for verification?

# Outline

1. Futures and its deadlocks
2. **Known Joins ⇒ DF**: Deadlock avoidance with futures & benchmarks
3. **DRF ⇒ Known Joins**: How DRF enjoys Deadlock-Freedom
4. Conclusion and future work

# 1. Futures and its deadlocks

# Futures: Tasks that "return" values

`async: (unit→T) → Future<T>`

- **Control:** Forks a task `A`
- **Data:** Returns the future value of type `Future<T>`

`get: Future<T> → T`

- **Control:** Joins with task `A`
- **Data:** Returns the value of type `T` "produced" by task `A`

# Deadlocked example

```
// Task P
shared Future<Integer> x, y;
x = async(() -> y.get()); // Task A
y = async(() -> x.get()); // Task B
```

1. `P` forks `A` writes to `x`
   - `A` waits for the task in `y`
2. `P` forks `B` writes to `y`
   - `B` waits for the task in `x`

**Data-race causes 2 traces**

# Trace 1 (no deadlock)

```
shared Future<Integer> x, y;
x = async(() -> y.get()); // y = null
y = async(() -> x.get());
```

| *Task P* | *Task A* | *Task B* |
|---|---|---|
| fork A | read y null | read x A |
| write x A | | get A |
| fork B | | |
| write y B | | |

# Trace 2 (deadlock)

```
shared Future<Integer> x, y;
x = async(() -> y.get()); // y = B
y = async(() -> x.get());
```

| *Task P* | *Task A* | *Task B* |
|---|---|---|
| `fork A` | `read y B` | `read x A` |
| `write x A` | `get B` | `get A` |
| `fork B` | | |
| `write y B` | | |

# Proving DRF ⇒ DF

DRF ⇒ $POLICY ⇒ DF

Deadlock-freedom policy valid in all DRF programs

# 2. Known Joins ⇒ DF

## Deadlock avoidance with futures & benchmarks

# Known-Joins implementation overview

## Program start (empty-known set)

**`async`**

1. Before: parent copies known-set to child
2. After: parent extends known-set with new task

**`get`**

1. Before: membership-test fail ⇒ POLICY ABORT
2. After: merge known-set of task

# Running example knowledge

Knowledge: {}

```
shared Future<Integer> x, y;
x = async(() -> y.get());
```

Knowledge: {A}

```
y = async(() -> x.get());
```

Knowledge: {A, B}

# Know-Joins in practice

## Habanero-Java: A Java 8 parallel programming library

### Extends the deadlock-free API subset with futures!

- `isolated`: mutual-exclusion
- `phaser`: barrier and producer-consumer
- `finish`: descendant task termination
- **`future` (with the known-joins policy)**

# Evaluation

- 2,300 assignments checked (1 unkown join, deadlocked example)
- 5 benchmarks

| ***Benchmark*** | ***# of async*** | ***# of get*** |
|---|---|---|
| Jacobi | 15,872 | 37,696 |
| Smith-Waterman | 21,000 | 4,641 |
| Crypt | 16,384 | 16,384 |
| Strassen | 30,811 | 44,816 |
| Series | 999,999 | 999,999 |

# Evaluation: time overhead

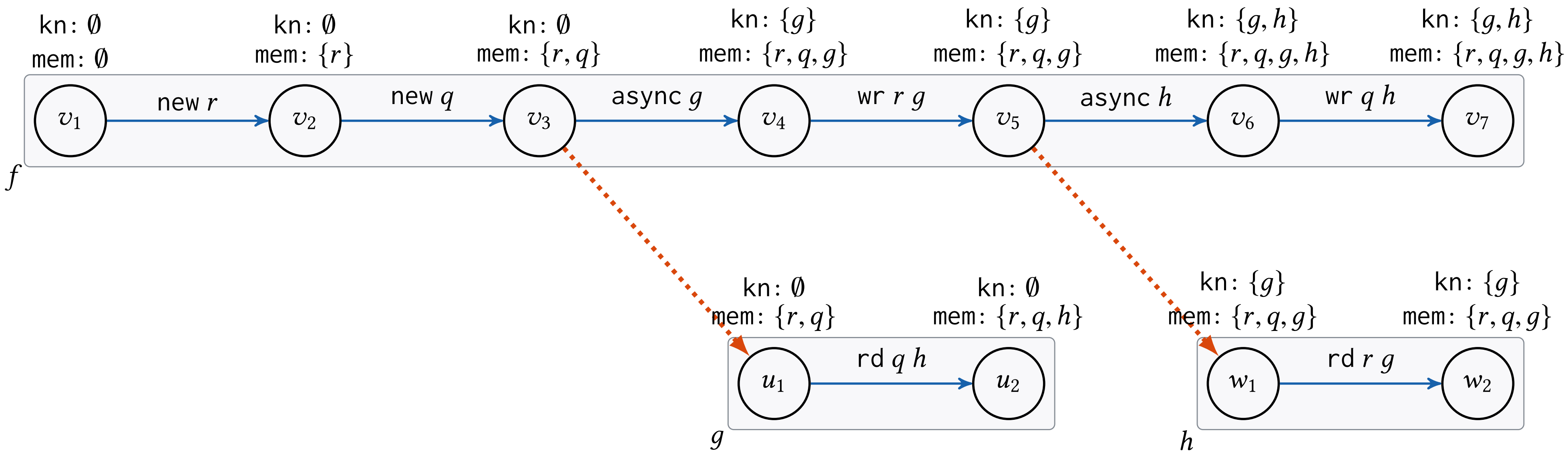
| ***Benchmark*** | ***Snapshot-sets*** |
|---|---|
| Jacobi | 0.99× |
| Smith-Waterman | 0.96× |
| Crypt | 1.04× |
| Strassen | 1.07× |
| Series | 1.06× |

# Evaluation: memory overhead

| *Benchmark* | *Snapshot-sets* |
|---|---|
| Jacobi | 1.00× |
| Smith-Waterman | 1.00× |
| Crypt | 1.08× |
| Strassen | 1.28× |
| Series | 2.34× |

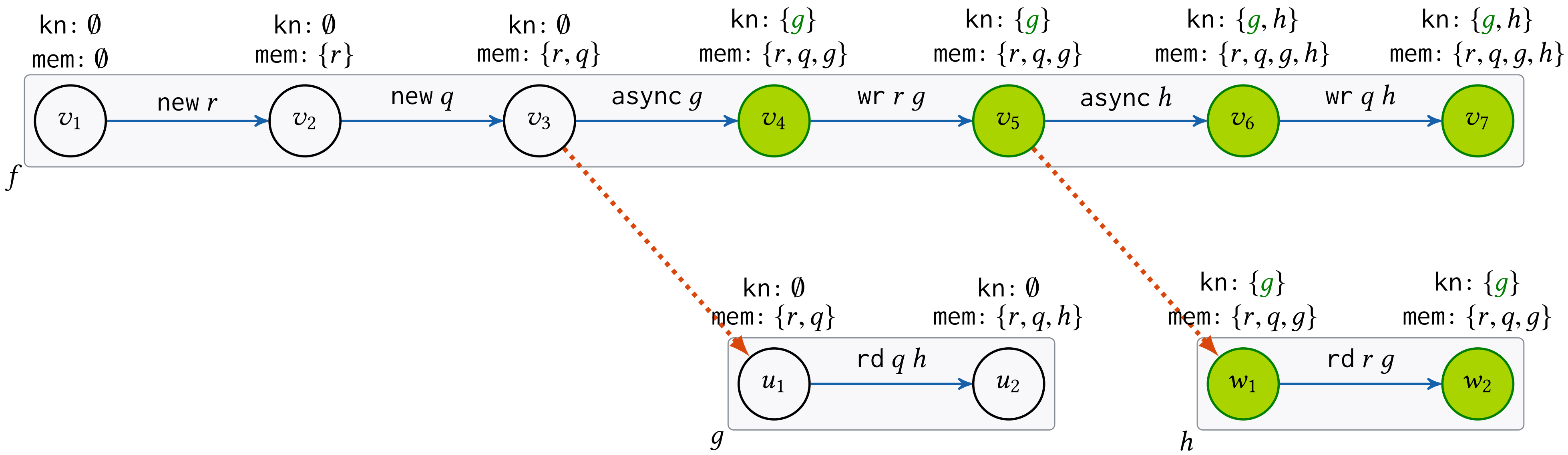
# 3. DRF ⇒ Known Joins

# Computation Graphs



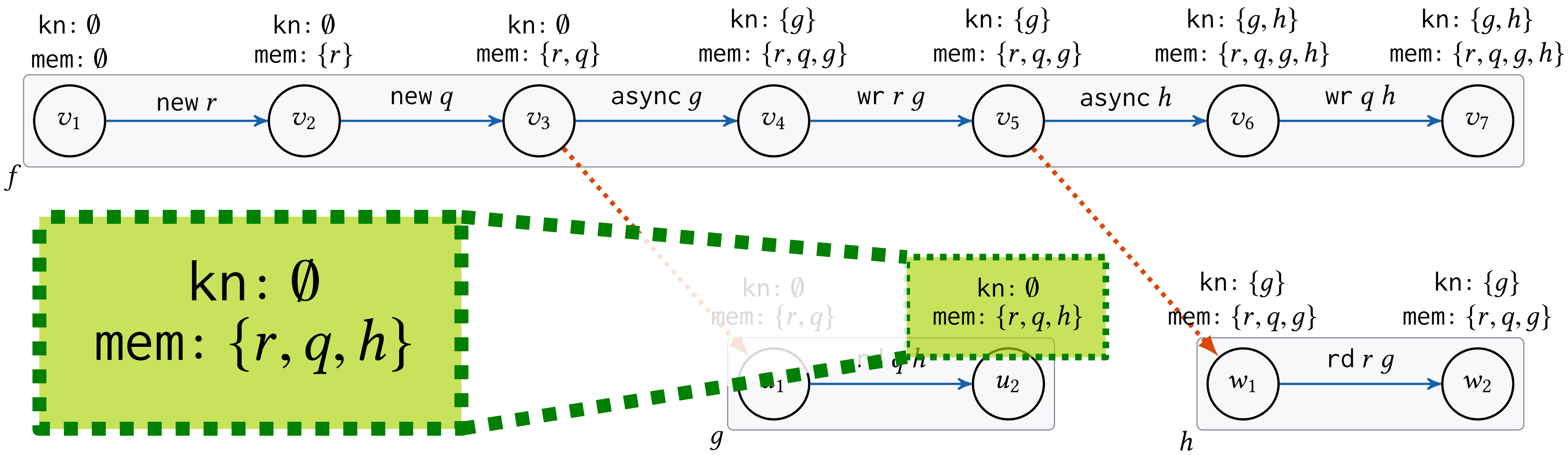
- Nodes: instruction instances
- Edges: happens-before dependencies (`async`, `get`, and sequential)
- Node annotations: **`kn`**own tasks and local **`mem`**ory

# Knowledge flows with reachability



```
v4 knows g
v4 happens-before w2
----------------------
w2 knows g
```

# Knowledge must contain tasks in memory

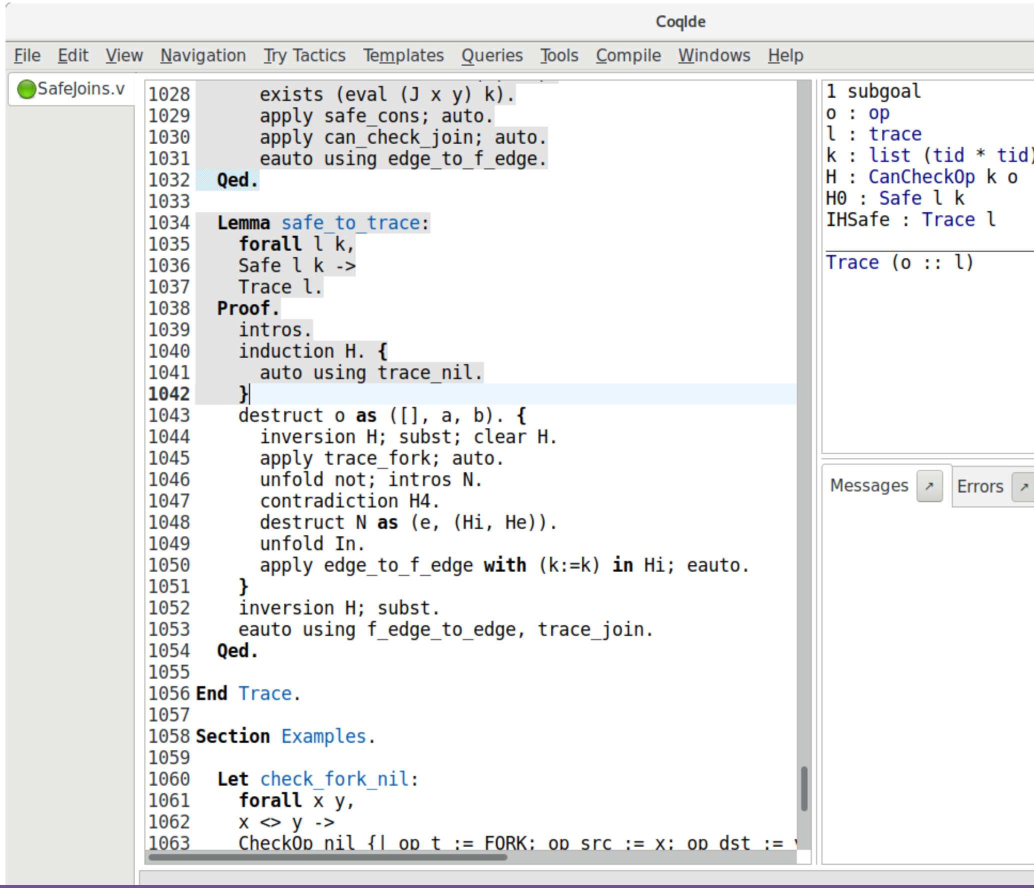


# In DRF graphs

# Main results

## Mechanized and proved in Coq

1. Known-Joins ⇒ Deadlock Freedom
2. Data-Race Freedom ⇒ Known-Joins
3. Know-Joins interpretation as a causality query



```
      exists (eval (J x y) k).
      apply safe_cons; auto.
      apply can_check_join; auto.
      eauto using edge_to_f_edge.
  Qed.

  Lemma safe_to_trace:
    forall l k,
    Safe l k ->
    Trace l.
  Proof.
    intros.
    induction H. {
      auto using trace_nil.
    }
    destruct o as ([], a, b). {
      inversion H; subst; clear H.
      apply trace_fork; auto.
      unfold not; intros N.
      contradiction H4.
      destruct N as (e, (Hi, He)).
      unfold In.
      apply edge_to_f_edge with (k:=k) in Hi; eauto.
    }
    inversion H; subst.
    eauto using f_edge_to_edge, trace_join.
  Qed.

End Trace.

Section Examples.

  Let check_fork_nil:
    forall x y,
    x <> y ->
    CheckOp nil {| op_t := FORK; op_src := x; op_dst :=
```

```
1 subgoal
o : op
l : trace
k : list (tid * tid)
H : CanCheckOp k o
H0 : Safe l k
IHSafe : Trace l
______________________________________
Trace (o :: l)
```

# Conclusion

- Introduced a theory of futures and shared memory (CG)
- Showed that data-races are the root cause of deadlocks
- Talked about a deadlock avoidance tool (1.06× time-overhead for 1 million tasks)

# Future work

- Promises lack runtime-information to derive deadlock detection
- Extend the theoretical framework for nondeterminism

# Deadlock Avoidance in Parallel Programs with Futures

## Why Parallel Tasks Should Not Wait for Strangers

Tiago Cogumbreiro, Rishi Surendran, Francisco Martins,
Vivek Sarkar, Vasco Vasconcelos, Max Grossman

OOPSLA, Vancouver, 2017