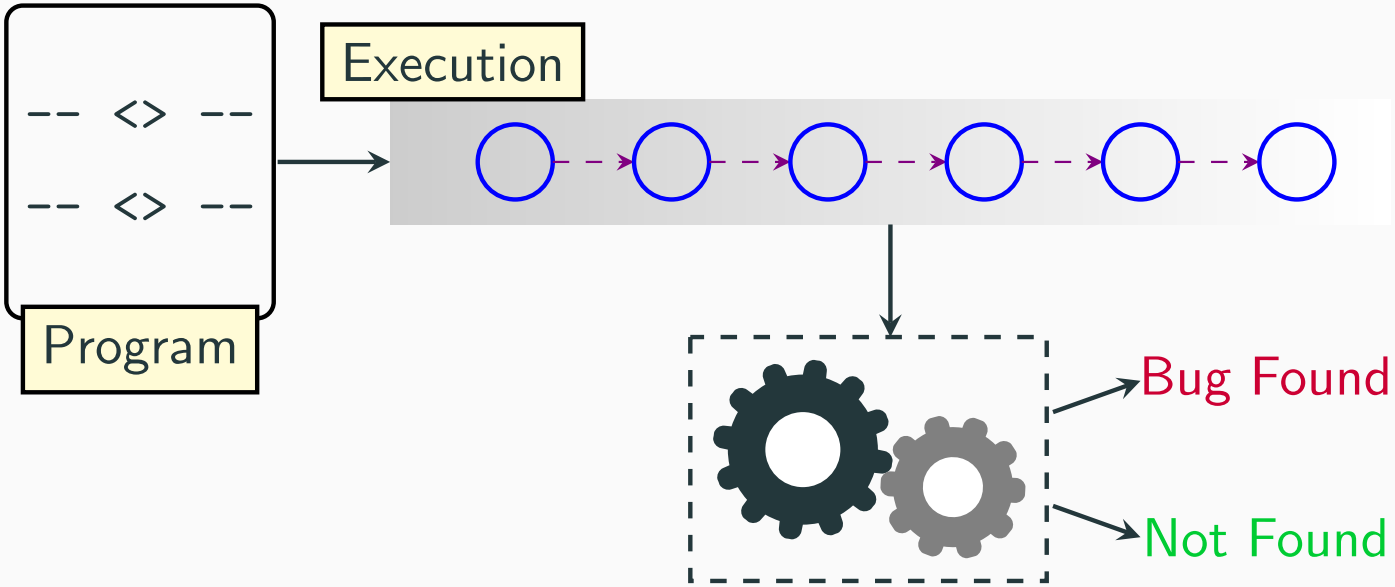
Concurrent Dynamic Analyses and the Tree Clock Data Structure

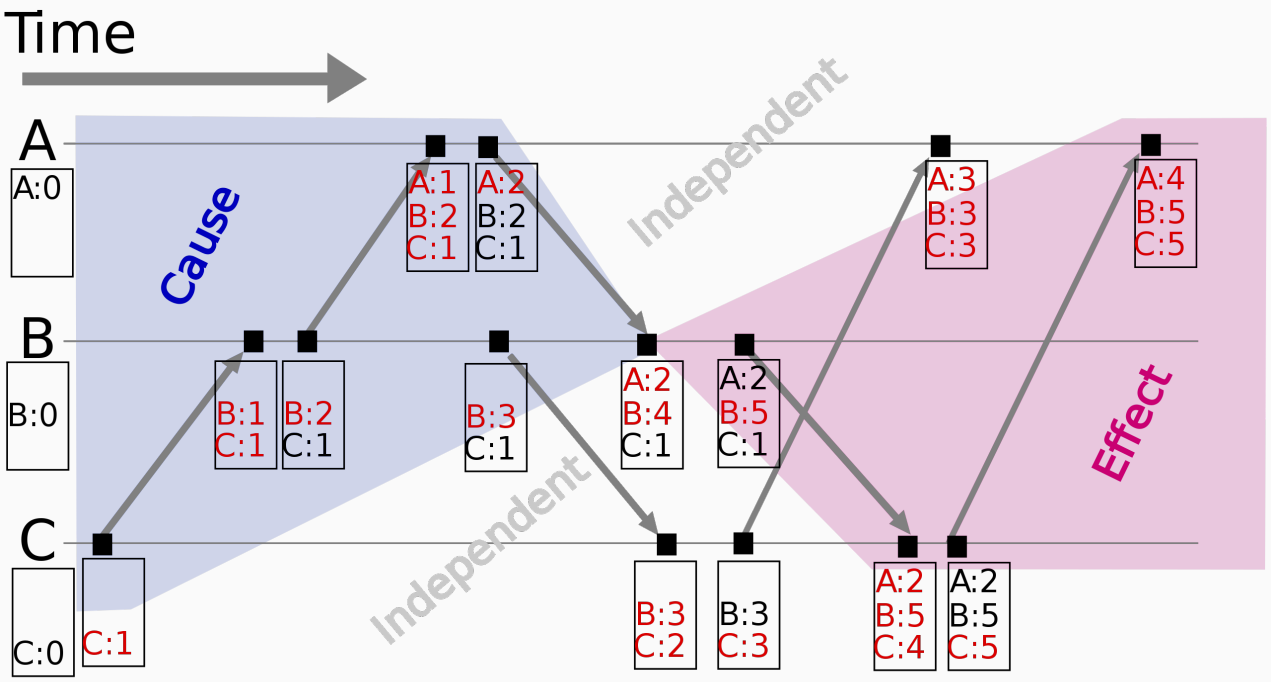
Andreas Pavlogiannis

AARHUS UNIVERSITY

Analyses of Concrete Concurrent Executions



Causality Analysis in Distributed Systems



Happens-Before (HB) Partial Order

For data-race detection over shared memory

	t_1	t_2
1	acquire(ℓ_1)	
2	*write*(x)	
3	release(ℓ_1)	
4		*read*(x)
5		acquire(ℓ_1)
6		*read*(x)
7		release(ℓ_1)

HB

Happens-Before (HB) Partial Order

For data-race detection over shared memory

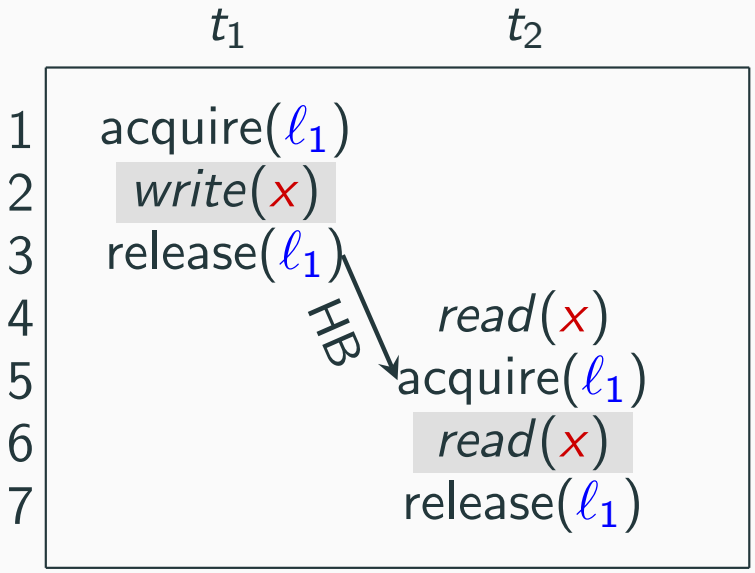
	t_1	t_2
1	acquire(ℓ_1)	
2	*write*(x)	
3	release(ℓ_1)	
4		*read*(x)
5		acquire(ℓ_1)
6		*read*(x)
7		release(ℓ_1)

HB

$\rightarrow$ Events e_2 and e_4 are concurrent

Happens-Before (HB) Partial Order

For data-race detection over shared memory



→ Events e_2 and e_4 are concurrent

→ Events e_2 and e_6 are **racy**

The Data-Race Detection Problem

Data-race detection

Given an input trace σ of size n over k threads, is there a data race in σ?

A data race is a pair

- $write(x), read(x)$, or
- $write(x), write(x)$

unordered by HB

The Data-Race Detection Problem

Data-race detection

Given an input trace σ of size n over k threads, is there a data race in σ?

A data race is a pair

- $write(x), read(x)$, or
- $write(x), write(x)$

unordered by HB

- Assume that for every event e, I know all events e' such that $e' \leq_{\mathsf{HB}} e$
- Then data-race detection is trivial, in $O(n)$ time

The Data-Race Detection Problem

Data-race detection

Given an input trace σ of size n over k threads, is there a data race in σ?

A data race is a pair

- $write(x), read(x)$, or
- $write(x), write(x)$

unordered by HB

- Assume that for every event e, I know all events e' such that $e' \leq_{\mathsf{HB}} e$
- Then data-race detection is trivial, in $O(n)$ time

- Computing HB takes $O(n \cdot k)$ time $\rightarrow$ quadratic!
- Using vector clocks

Vector Timestamps

- Succinct representation of the knowledge set of a thread t:

$$\mathsf{V}_t : \text{ Threads} \to \mathbb{N}$$

- $\mathsf{V}_t(t')$ gives the last event of t' that t knows about.
- t knows about all preceding events as well.

Vector Timestamps

- Succinct representation of the knowledge set of a thread t:

$$\mathsf{V}_t : \text{ Threads} \to \mathbb{N}$$

- $\mathsf{V}_t(t')$ gives the last event of t' that t knows about.
- t knows about all preceding events as well.

$$\mathsf{V}_{t_2} = [\overset{t_1}{27}, \overset{t_2}{3}, \overset{t_3}{9}, \overset{t_4}{45}, \overset{t_5}{17}, \overset{t_6}{26}]$$

- t_2 knows of the first 27 events of t_1.
- t_2 has performed 3 events.

Vector Timestamps

- Succinct representation of the knowledge set of a thread t:

$$\mathsf{V}_t\colon\ \mathsf{Threads} \to \mathbb{N}$$

- $\mathsf{V}_t(t')$ gives the last event of t' that t knows about.
- t knows about all preceding events as well.

$$\mathsf{V}_{t_2} = [\overset{t_1}{27}, \overset{t_2}{3}, \overset{t_3}{9}, \overset{t_4}{45}, \overset{t_5}{17}, \overset{t_6}{26}]$$

- t_2 knows of the first 27 events of t_1.
- t_2 has performed 3 events.

<u>Operations</u>

$$\begin{array}{lcll} \mathsf{V}_1 \sqsubseteq \mathsf{V}_2 & \text{iff} & \forall t\colon\ \mathsf{V}_1(t) \leq \mathsf{V}_2(t) & \text{(Comparison)} \\ \mathsf{V}_1 \sqcup \mathsf{V}_2 & = & \lambda t\colon\ \max(\mathsf{V}_1(t), \mathsf{V}_2(t)) & \text{(Join)} \end{array}$$

Implementing Vector Timestamps

Just use a vector clock $\mathrm{VC}_t = [27, 3, 9, 45, 17, 26]$

Vector Clock Join $\mathrm{VC}_1 \leftarrow \mathrm{VC}_1 \sqcup \mathrm{VC}_2$

- For each thread t:
  - If $\mathrm{VC}_1[t] < \mathrm{VC}_2[t]$
    - $\mathrm{VC}_1[t] \leftarrow \mathrm{VC}_2[t]$

Vector Clock Copy $\mathrm{VC}_1 \leftarrow \mathrm{VC}_2$

- For each thread t:
  - $\mathrm{VC}_1[t] \leftarrow \mathrm{VC}_2[t]$

Each operation takes $O(k)$ time, for k threads

Computing Happens-Before with Vector Clocks

- One vector clock $\mathbb{C}_t$ per thread t
- One vector clock $\mathbb{C}_\ell$ per lock ℓ

Algorithm: Happens-Before (HB)

```
procedure acquire(t, ℓ)
    C_t ← C_t ⊔ C_ℓ;                    /* Vector clock join */
procedure release(t, ℓ)
    C_ℓ = C_t;                          /* Vector clock copy */
```

Computing Happens-Before with Vector Clocks

- One vector clock $\mathbb{C}_t$ per thread t
- One vector clock $\mathbb{C}_\ell$ per lock ℓ

Algorithm: Happens-Before (HB)

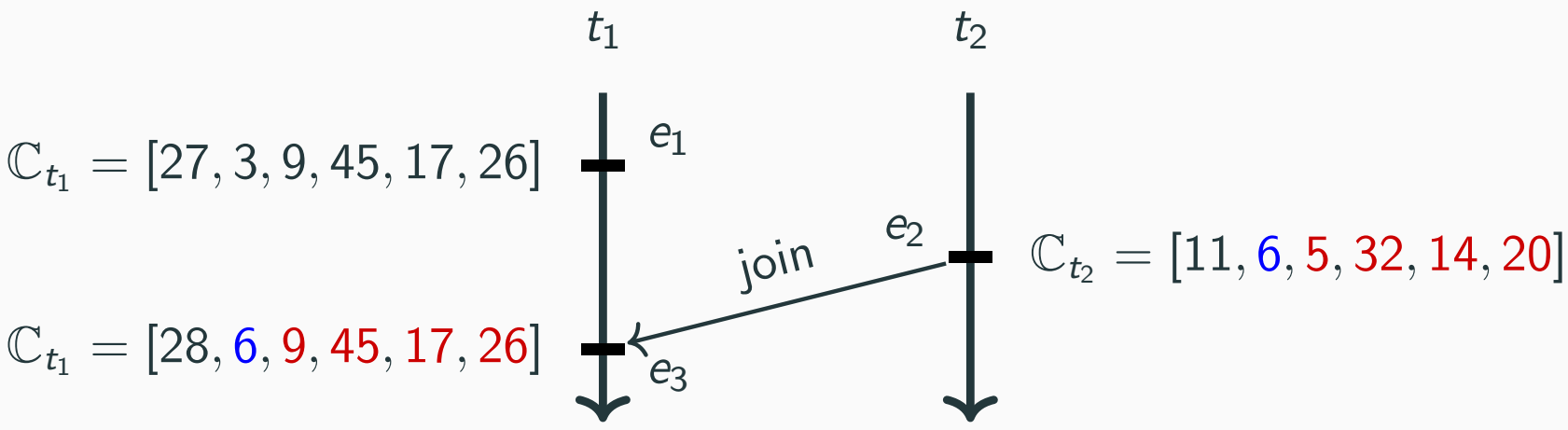
```
procedure acquire(t, ℓ)
    C_t ← C_t ⊔ C_ℓ;                    /* Vector clock join */
procedure release(t, ℓ)
    C_ℓ = C_t;                          /* Vector clock copy */
```

- Every vector clock operation costs $O(k)$
  - k is the number of threads
- When threads are many, the complexity is quadratic $\Theta(n \cdot k) = \Theta(n^2)$
  - n is the number of acquire/release events

Can we do better? $O(n)$?

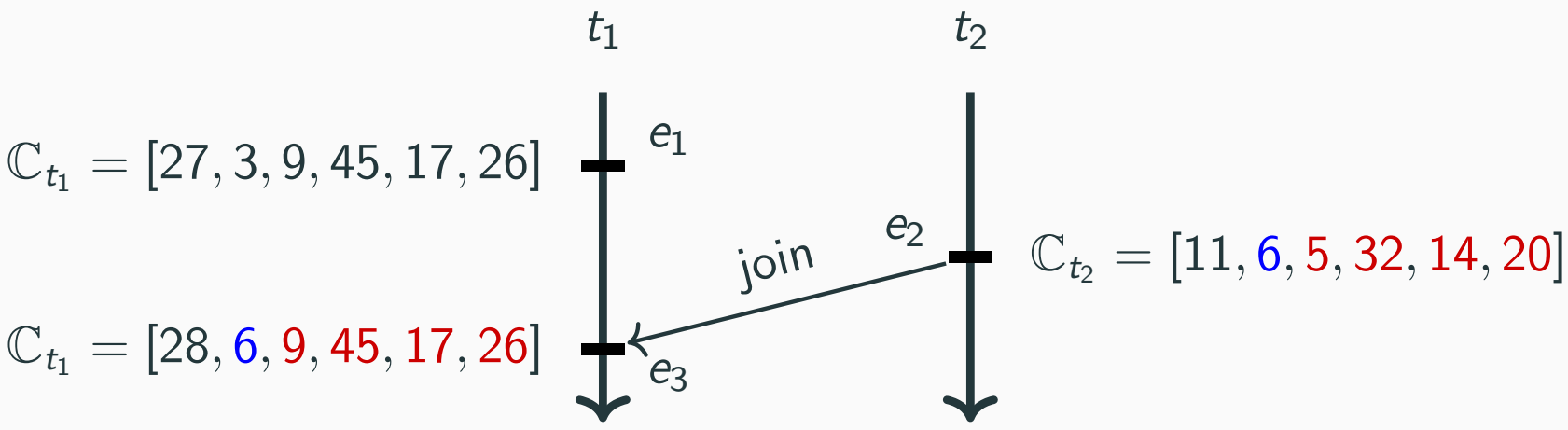
Overhead of Vector Clocks

- Every vector clock join takes $O(k)$ time.
- Certain steps in the join operation can be vacuous.



Overhead of Vector Clocks

- Every vector clock join takes $O(k)$ time.
- Certain steps in the join operation can be vacuous.



- In practice, many vacuous operations
- Perhaps a smarter algorithm for HB?
- Can we get complexity down to $O(n)$?

Fine-Grained Complexity

- Attempts to improve over the quadratic bound $O(n \cdot k)$ have failed
- How do we prove that a linear bound $O(n)$ is impossible/unlikely?

Fine-Grained Complexity

- Attempts to improve over the quadratic bound $O(n \cdot k)$ have failed
- How do we prove that a linear bound $O(n)$ is impossible/unlikely?

Fine-grained complexity theory

- Studies complexity relationships between problems in **P**
- "Problem A cannot be solved in $O(n^{2-\epsilon})$ time, assuming problem B cannot be solved in $O(n^{3-\delta})$ time"
- Separations like **P** vs **NP**, but with emphasis on the polynomial degree

The Fine-Grained Complexity of Race Detection

Dynamic Data-Race Detection Through the Fine-Grained Lens

Rucha Kulkarni
University of Illinois at Urbana-Champaign, IL, USA
Umang Mathur
University of Illinois at Urbana-Champaign, IL, USA
Andreas Pavlogiannis
Aarhus University, Denmark

The Fine-Grained Complexity of Race Detection

Dynamic Data-Race Detection Through the Fine-Grained Lens

Rucha Kulkarni
University of Illinois at Urbana-Champaign, IL, USA
Umang Mathur
University of Illinois at Urbana-Champaign, IL, USA
Andreas Pavlogiannis
Aarhus University, Denmark

Theorem

Detecting a single data race cannot take $O(n^{3/2-\epsilon})$ time on a trace of size n.

The Fine-Grained Complexity of Race Detection

Dynamic Data-Race Detection Through the Fine-Grained Lens

Rucha Kulkarni
University of Illinois at Urbana-Champaign, IL, USA
Umang Mathur
University of Illinois at Urbana-Champaign, IL, USA
Andreas Pavlogiannis
Aarhus University, Denmark

Theorem

Detecting a single data race cannot take $O(n^{3/2-\epsilon})$ time on a trace of size n.

Theorem

Detecting a single write-read data race cannot take $O(n^{2-\epsilon})$ time on a trace of size n.

The Fine-Grained Complexity of Race Detection

Dynamic Data-Race Detection Through the Fine-Grained Lens

Rucha Kulkarni
University of Illinois at Urbana-Champaign, IL, USA
Umang Mathur
University of Illinois at Urbana-Champaign, IL, USA
Andreas Pavlogiannis
Aarhus University, Denmark

Theorem

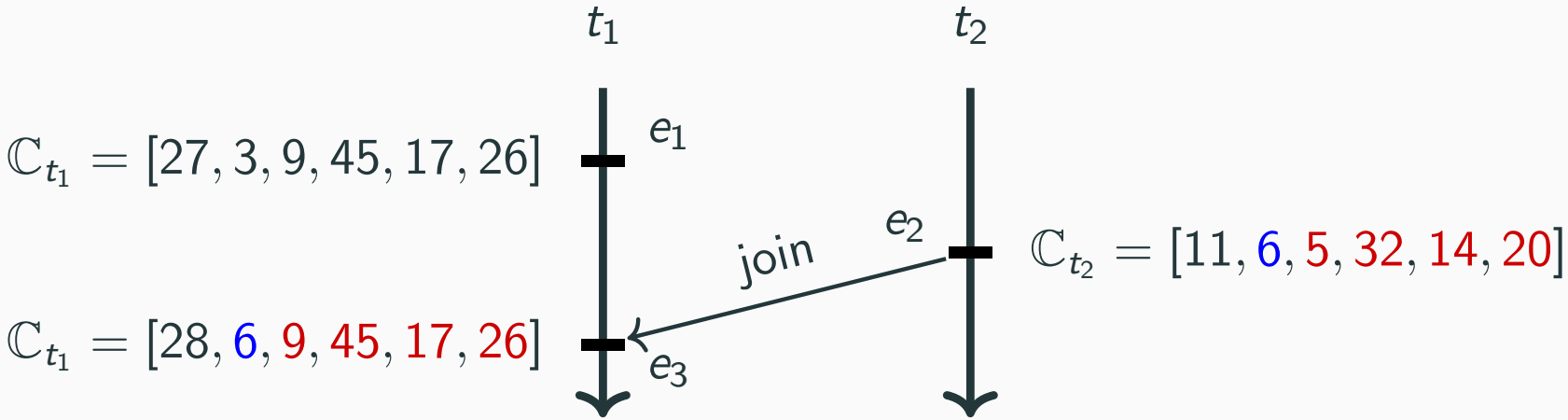
Detecting a single data race cannot take $O(n^{3/2-\epsilon})$ time on a trace of size n.

Theorem

Detecting a single write-read data race cannot take $O(n^{2-\epsilon})$ time on a trace of size n.

. . . under some fine-grained complexity conjectures

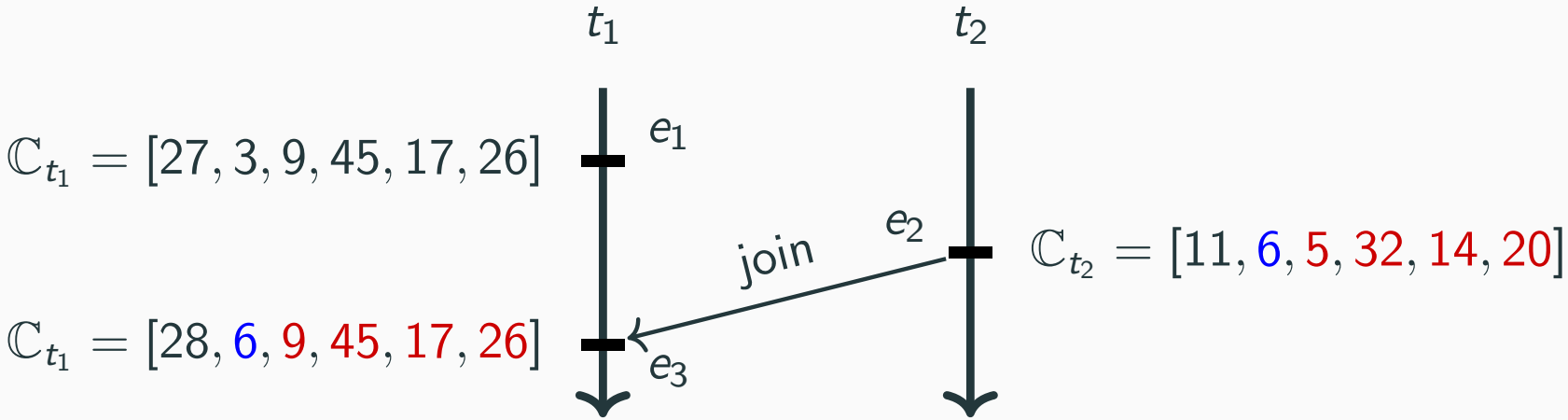
A More Pragmatic Approach



Can we do joins in sub-linear time?

$\rightarrow$ Sub-linear means skip looking at certain entries. How?

A More Pragmatic Approach



Can we do joins in sub-linear time?

→ Sub-linear means skip looking at certain entries. How?

A Tree Clock Data Structure for Causal Orderings in Concurrent Executions

Umang Mathur
National University of Singapore
Singapore
umathur@comp.nus.edu.sg

Andreas Pavlogiannis
Aarhus University
Denmark
pavlogiannis@cs.au.dk

Hünkar Can Tunç
Aarhus University
Denmark
tunc@cs.au.dk

Mahesh Viswanathan
University of Illinois at Urbana-Champaign
USA
vmahesh@illinois.edu

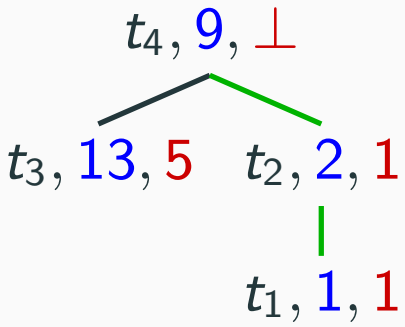
The Tree Clock Data Structure

- **Drop-in** replacement of vector clocks.
- Tree clocks maintain information hierarchically.
  - Nodes store local times of a thread + metadata.
  - Tree structure records the provenance of causality.

The Tree Clock Data Structure

- **Drop-in** replacement of vector clocks.
- Tree clocks maintain information hierarchically.
  - Nodes store local times of a thread + metadata.
  - Tree structure records the provenance of causality.

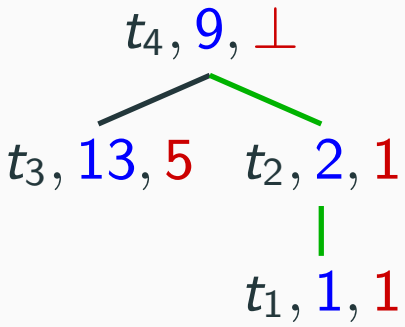
$$\mathbb{C}_{t_4} = [\overset{t_1}{1}, \overset{t_2}{2}, \overset{t_3}{13}, \overset{t_4}{9}]$$



The Tree Clock Data Structure

- **Drop-in** replacement of vector clocks.
- Tree clocks maintain information hierarchically.
  - Nodes store local times of a thread + metadata.
  - Tree structure records the provenance of causality.

$$\mathbb{C}_{t_4} = [\overset{t_1}{1}, \overset{t_2}{2}, \overset{t_3}{13}, \overset{t_4}{9}]$$

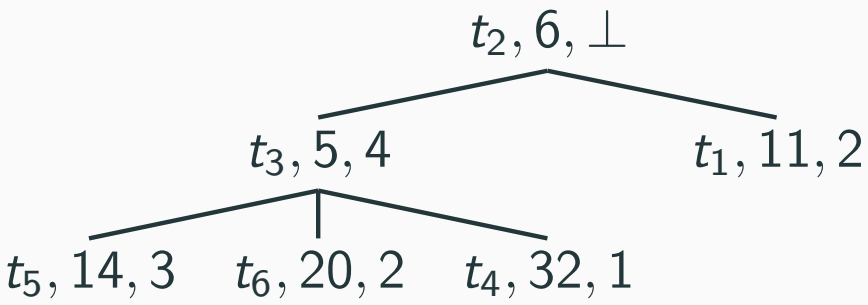


- Only slightly more information is stored compared to vector clocks.

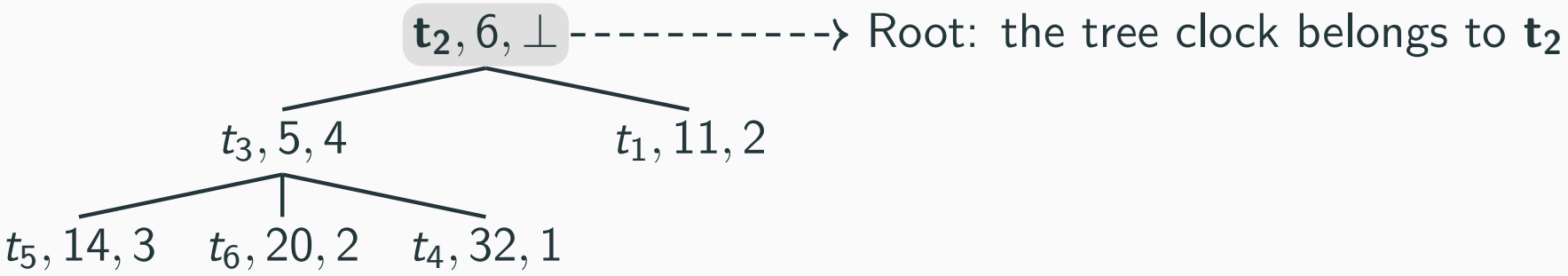
$\downarrow$

Enough to support joins in sub-linear time!

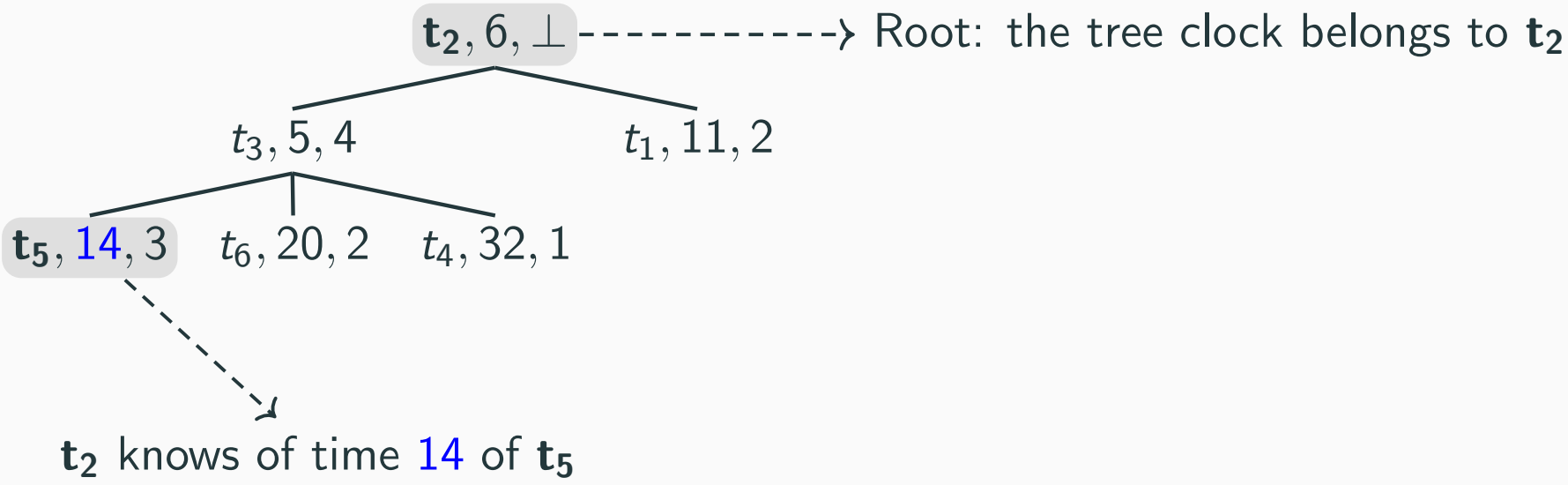
Tree Clock Data Structure



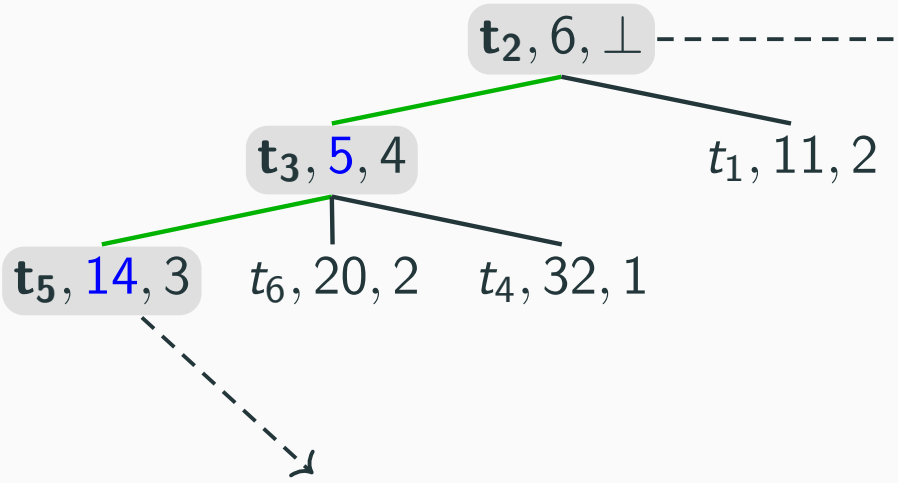
Tree Clock Data Structure



Tree Clock Data Structure



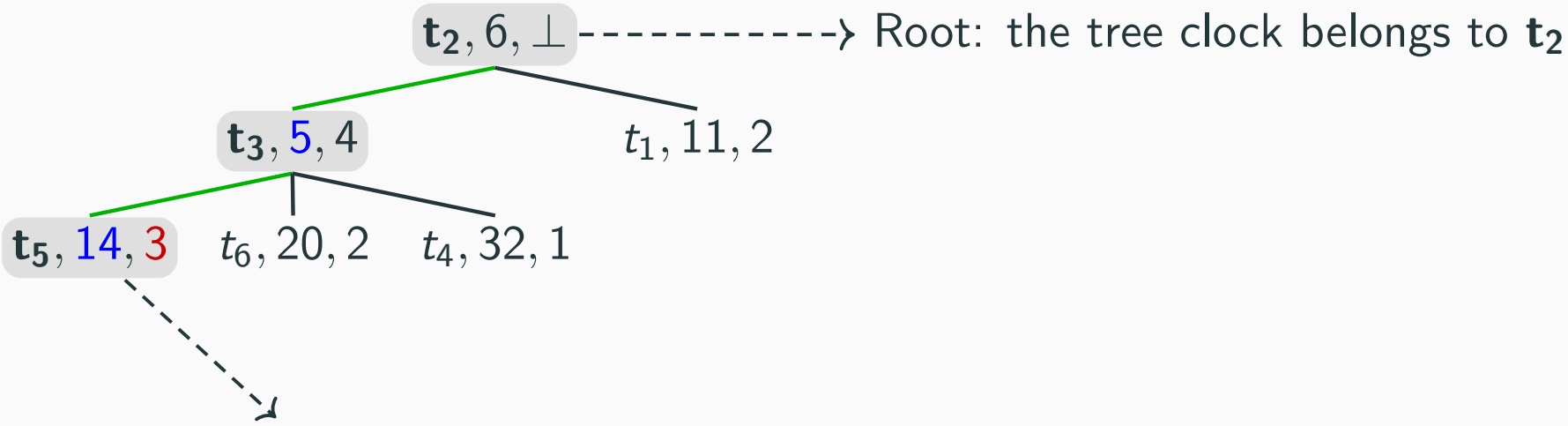
Tree Clock Data Structure



$\mathbf{t_2}$ knows of time 14 of $\mathbf{t_5}$

$\hookrightarrow$ It learned this transitively, by learning of time 5 of $\mathbf{t_3}$

Tree Clock Data Structure

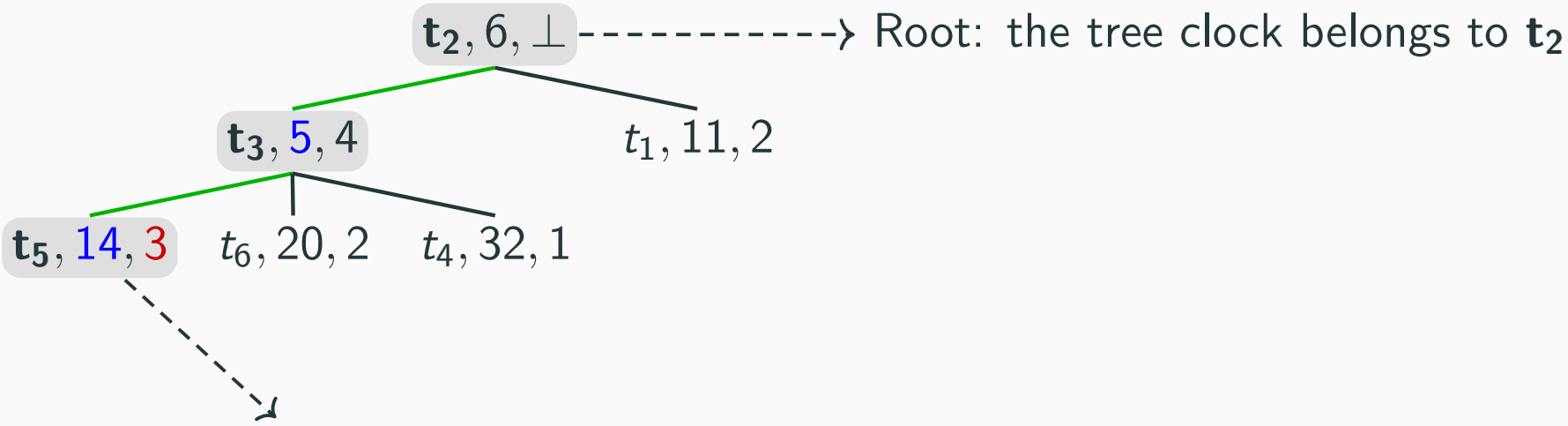


$\mathbf{t_2}$ knows of time 14 of $\mathbf{t_5}$

$\hookrightarrow$ It learned this transitively, by learning of time 5 of $\mathbf{t_3}$

$\hookrightarrow$ $\mathbf{t_3}$ learned of time 14 of $\mathbf{t_5}$ when $\mathbf{t_3}$'s time was 3

Tree Clock Data Structure



$\mathbf{t_2}$ knows of time 14 of $\mathbf{t_5}$

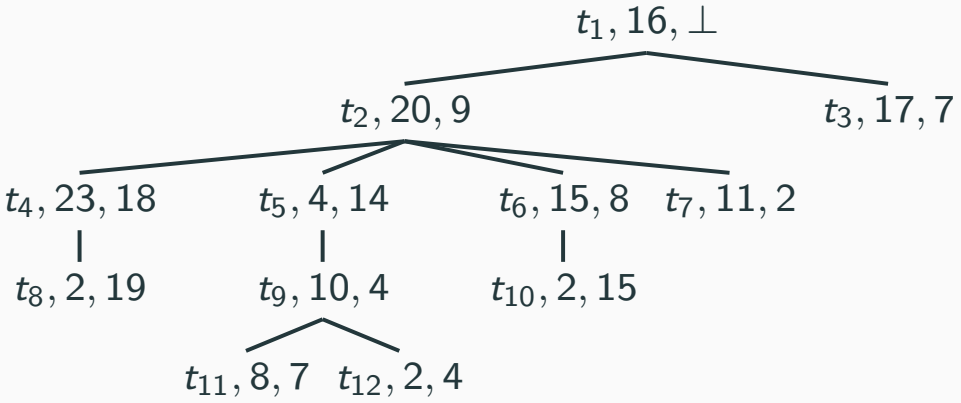
$\hookrightarrow$ It learned this transitively, by learning of time 5 of $\mathbf{t_3}$

$\hookrightarrow$ $\mathbf{t_3}$ learned of time 14 of $\mathbf{t_5}$ when $\mathbf{t_3}$'s time was 3

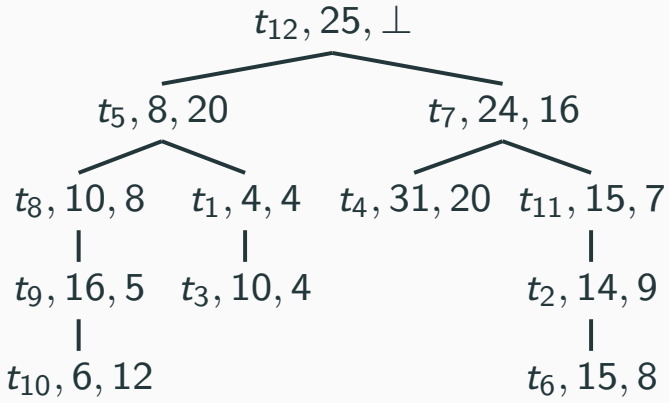
This structure allows for sub-linear time joins.

TC_{12} .Join(TC_1)

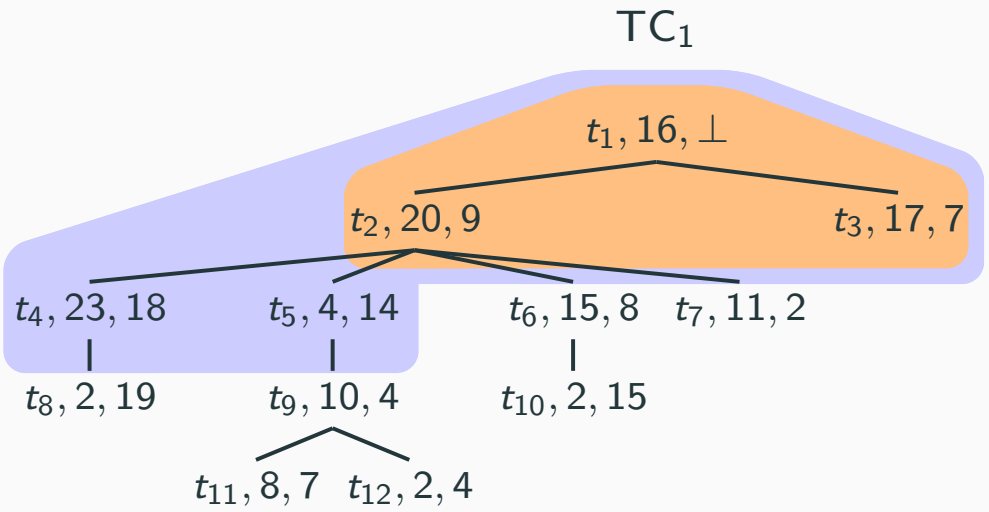
TC_1

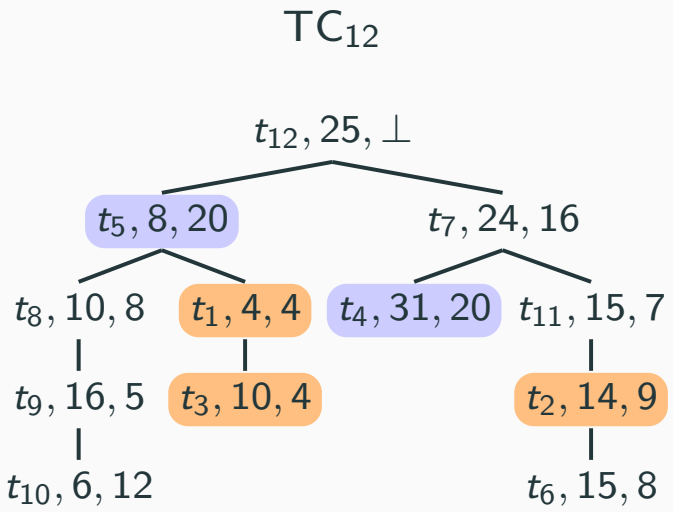


TC_{12}



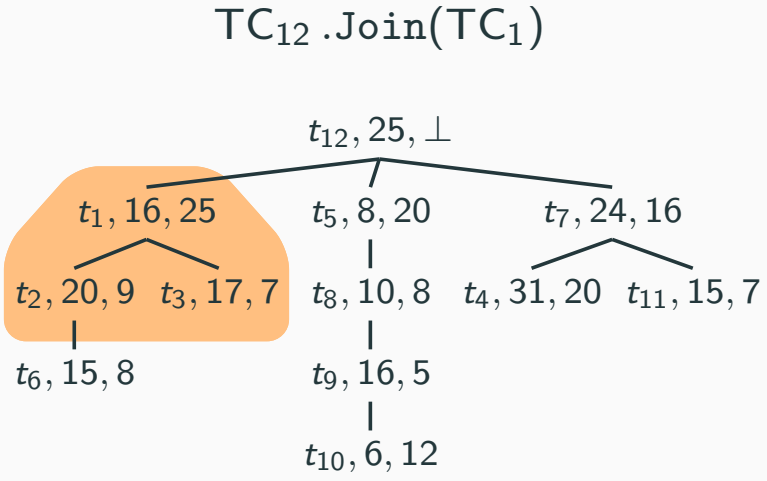
TC_{12} .Join(TC_1)



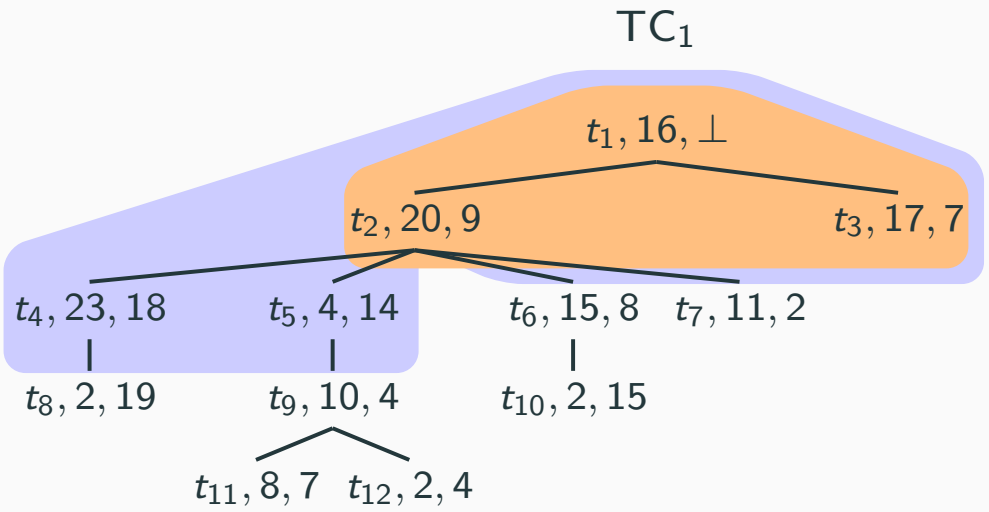


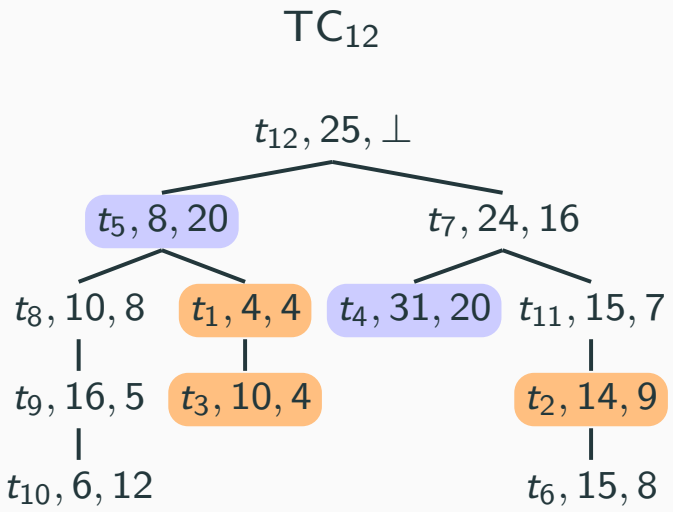
Updated

Accessed



TC_{12} .Join(TC_1)

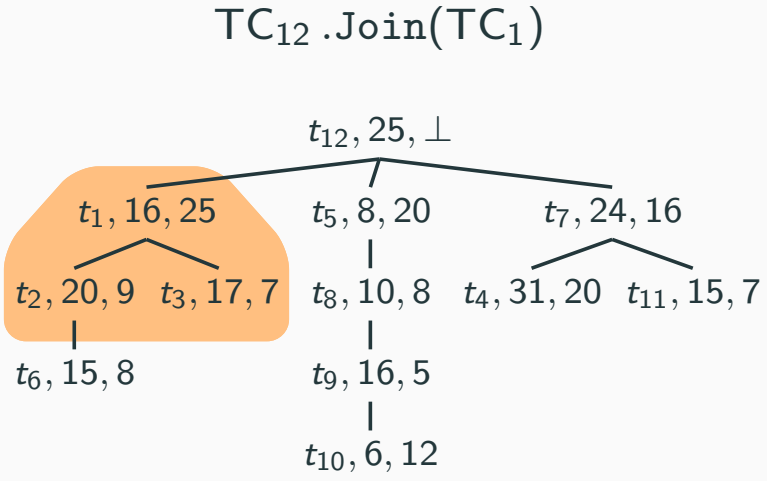




Updated

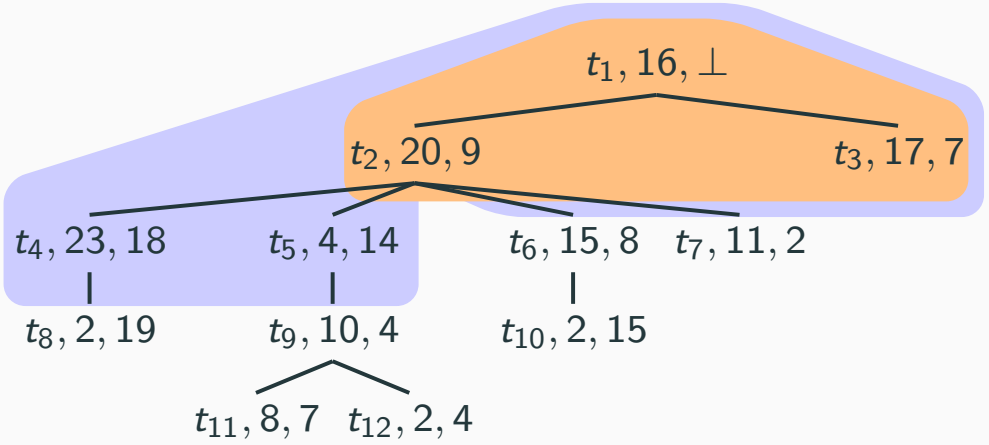
Accessed

Performed join without accessing the whole tree!

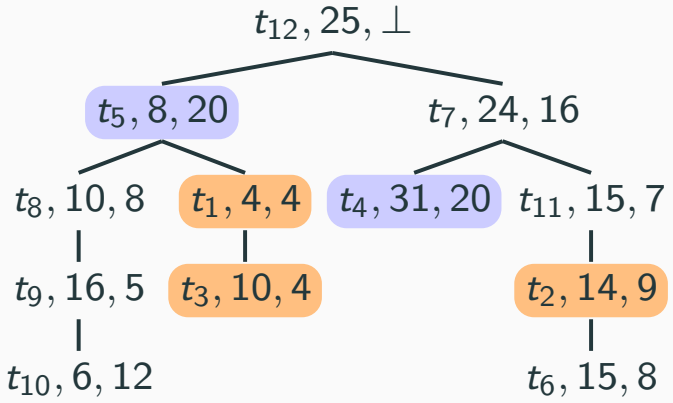


TC_{12} .Join(TC_1)

TC_1



TC_{12}

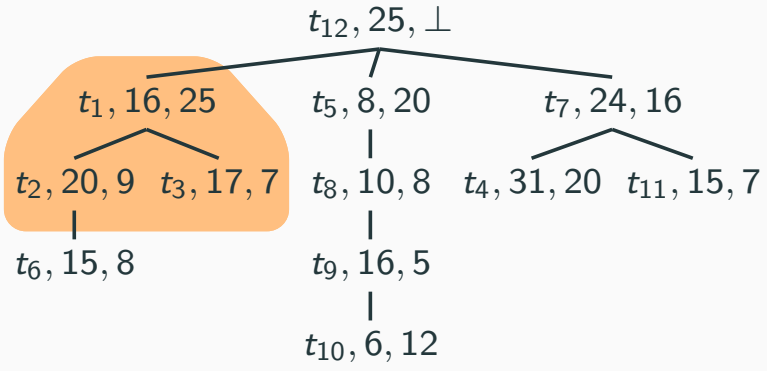


Updated

Accessed

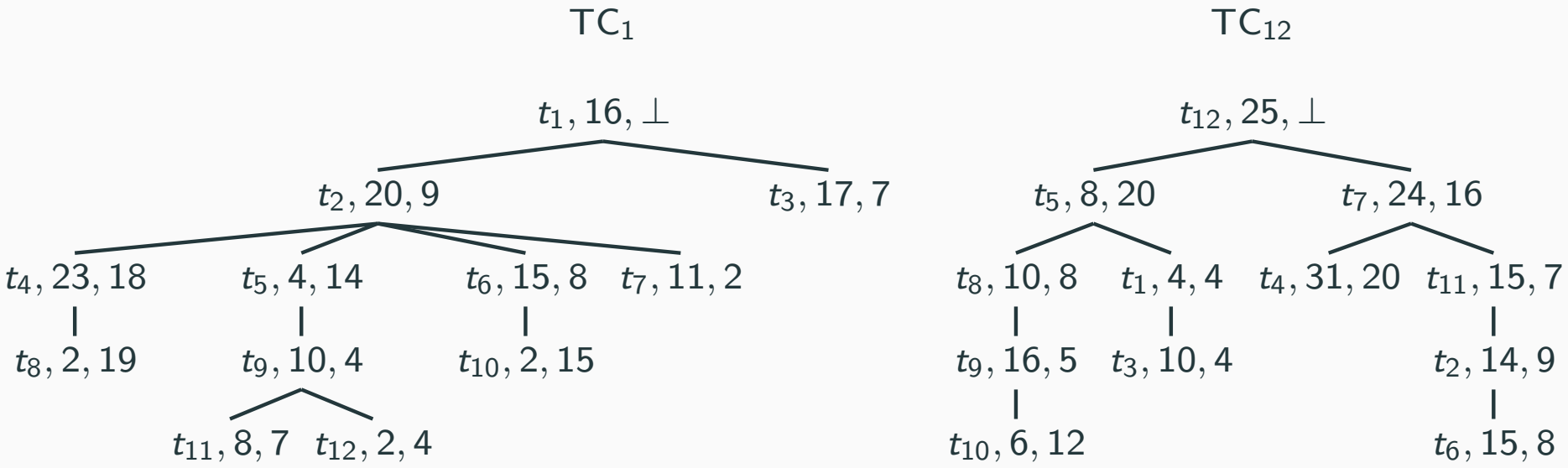
Performed join without accessing the whole tree!

TC_{12} .Join(TC_1)

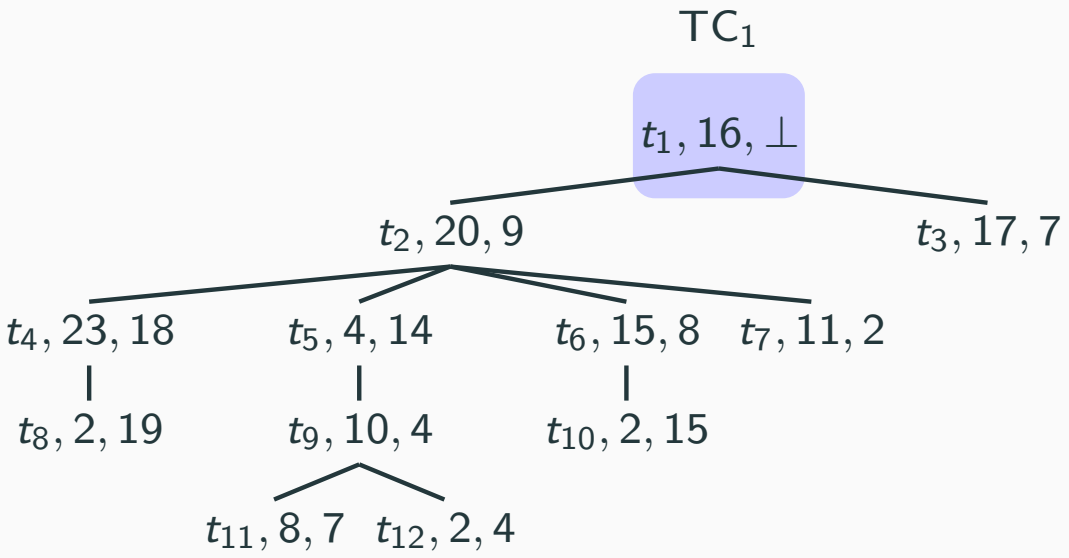


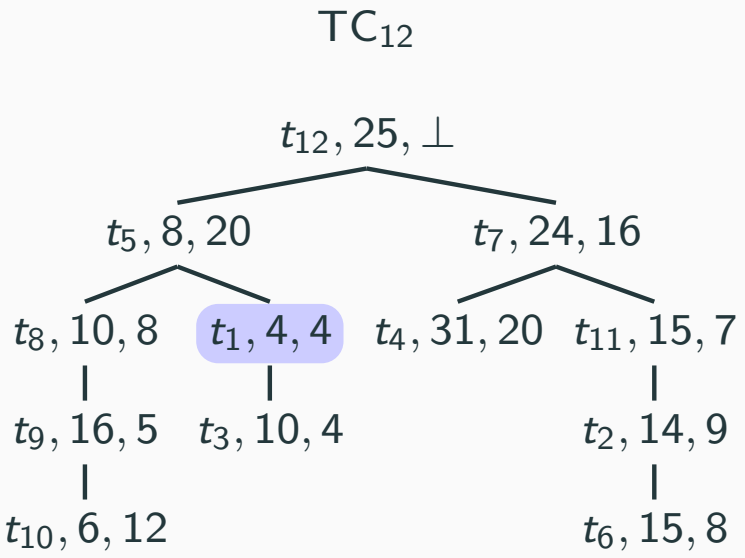
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
$\mathbb{C}_{t_1}$ =	16	20	17	23	4	15	11	8	10	2	8	7

TC_{12} .Join(TC_1)

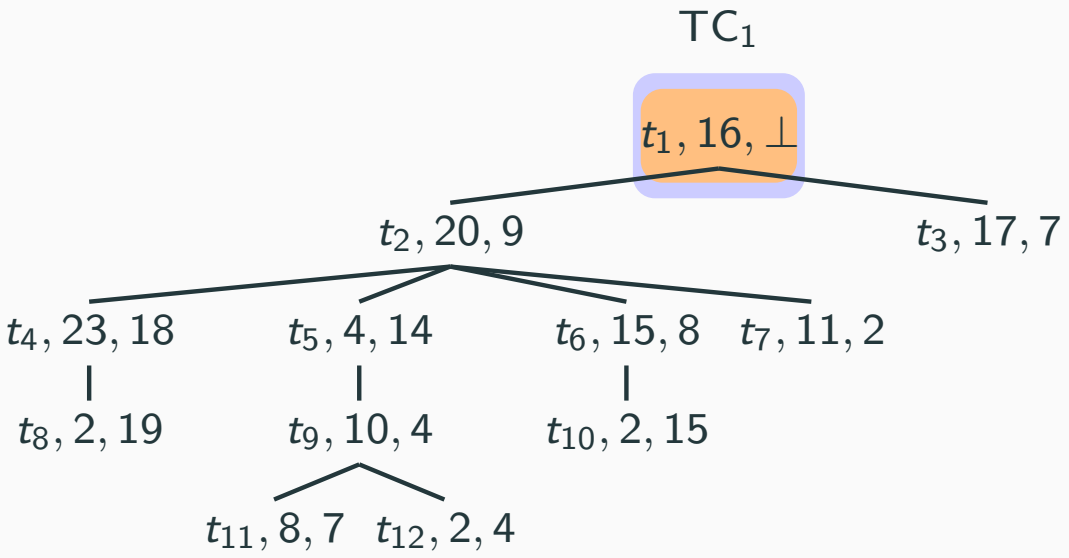


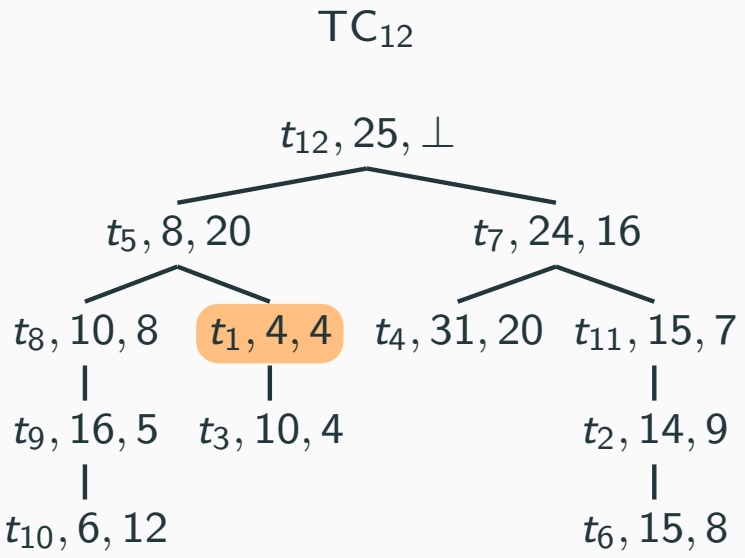
TC_{12} .Join(TC_1)



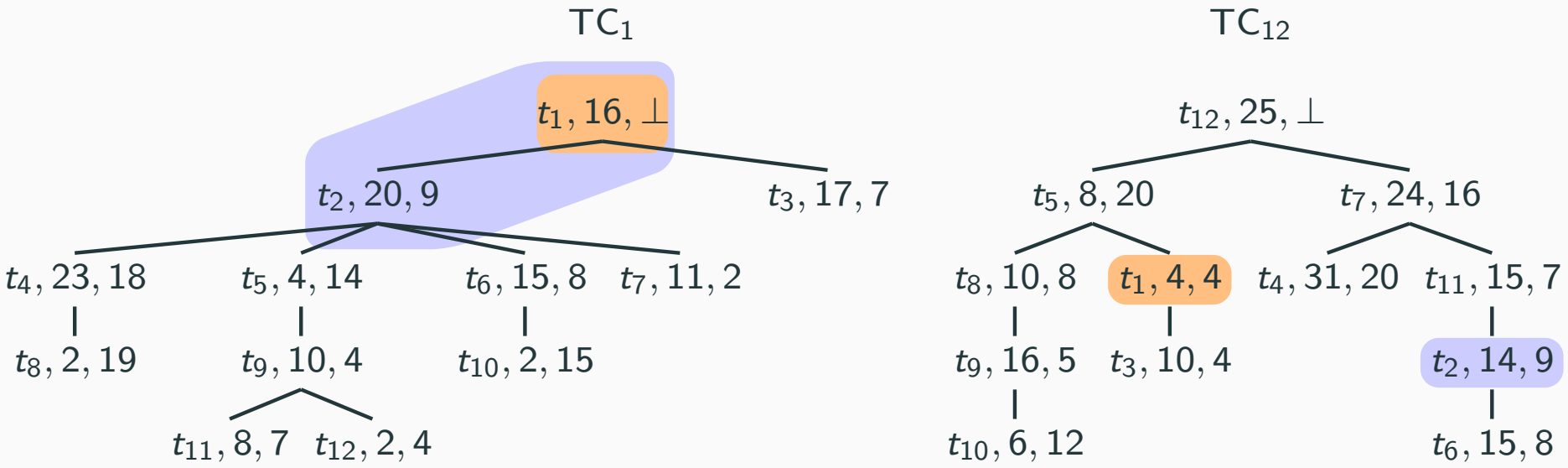


TC_{12} .Join(TC_1)

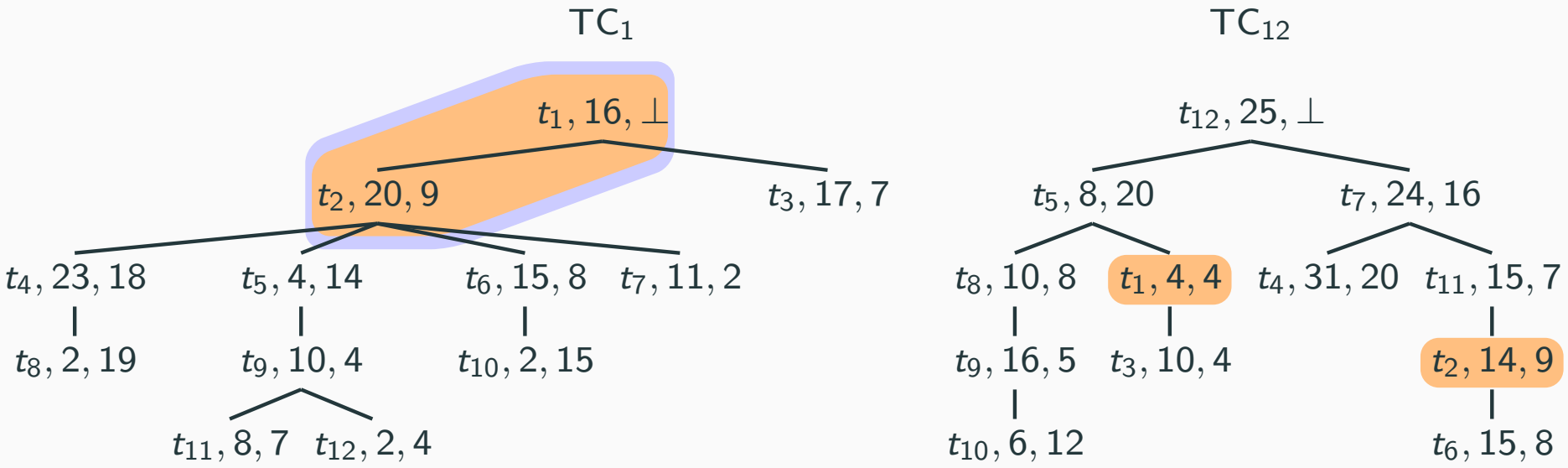




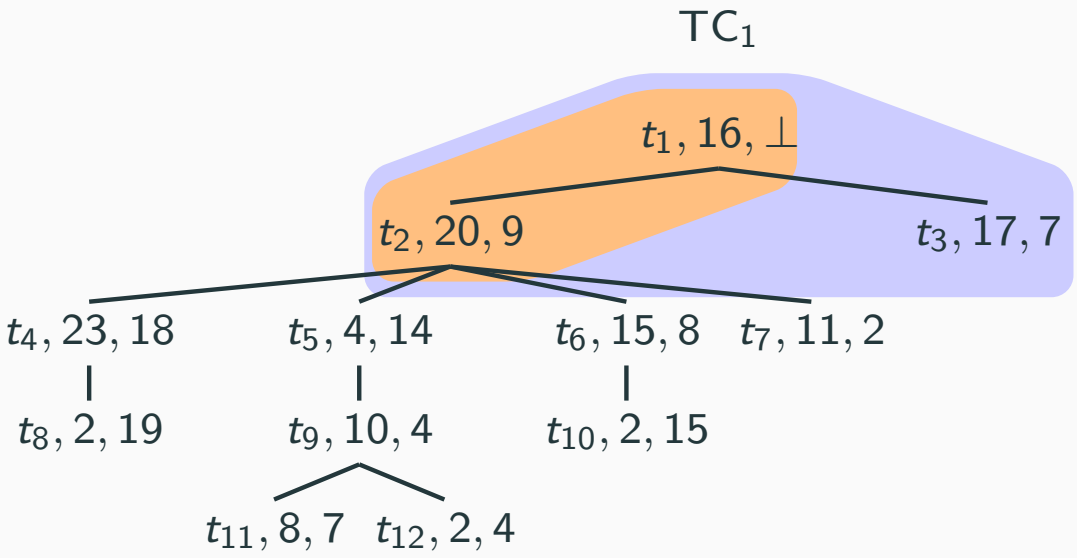
TC_{12} .Join(TC_1)

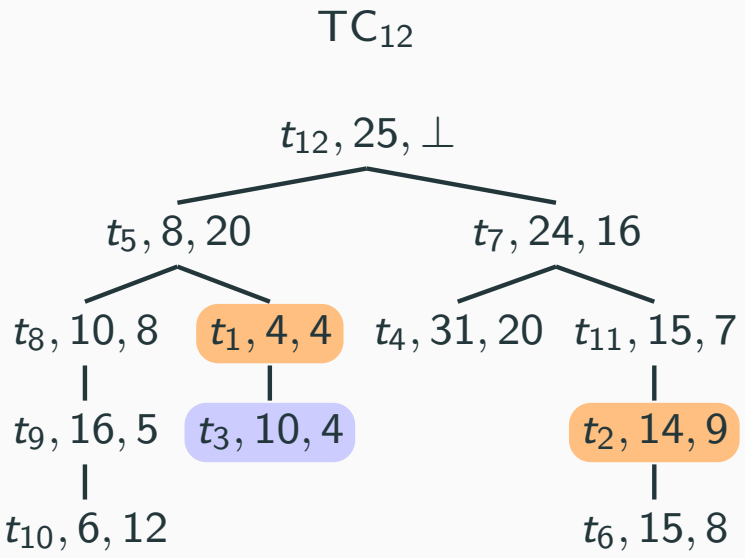


TC_{12} .Join(TC_1)

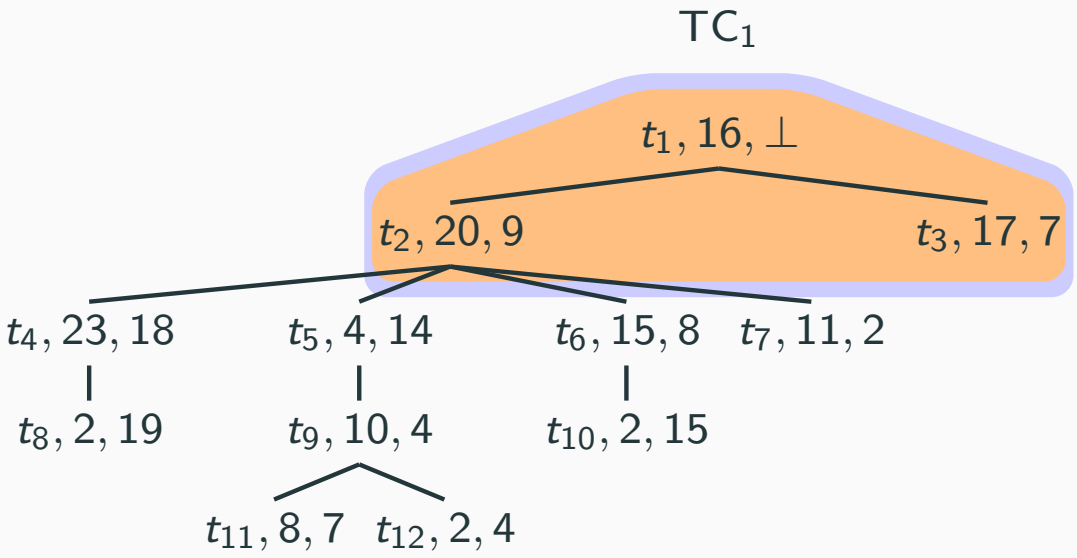


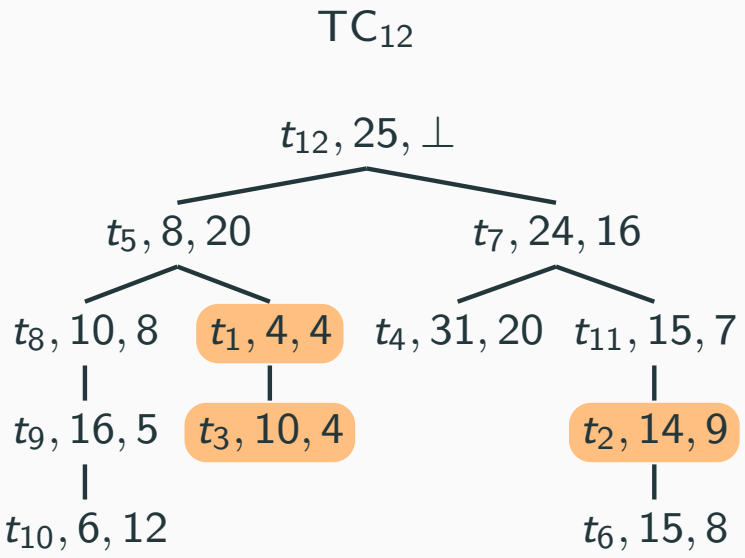
TC_{12} .Join(TC_1)



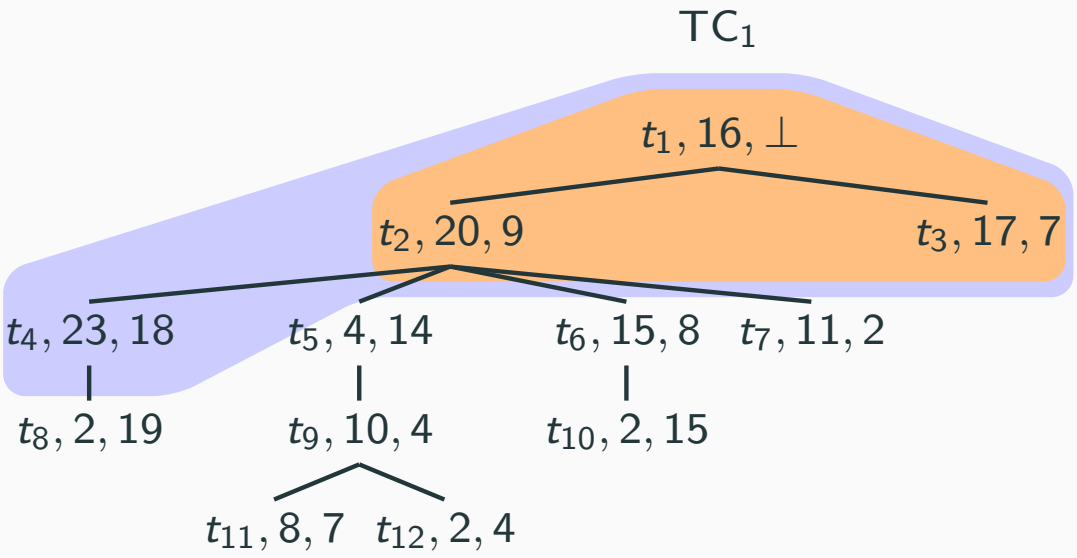


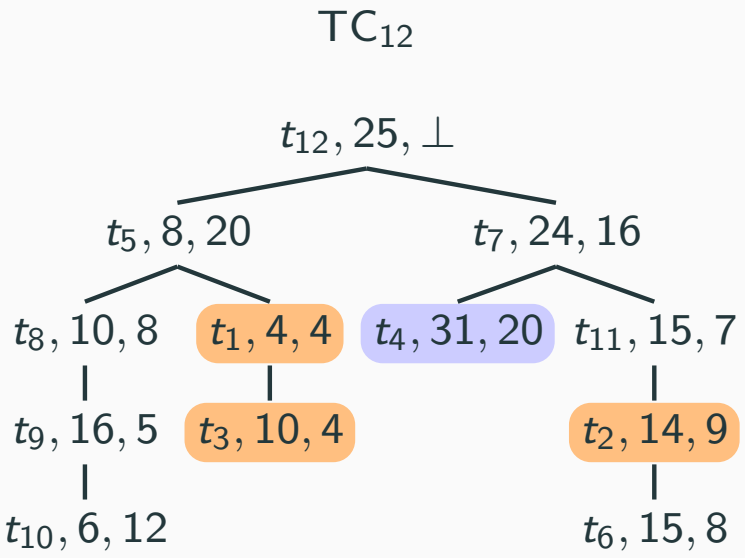
TC_{12} .Join(TC_1)



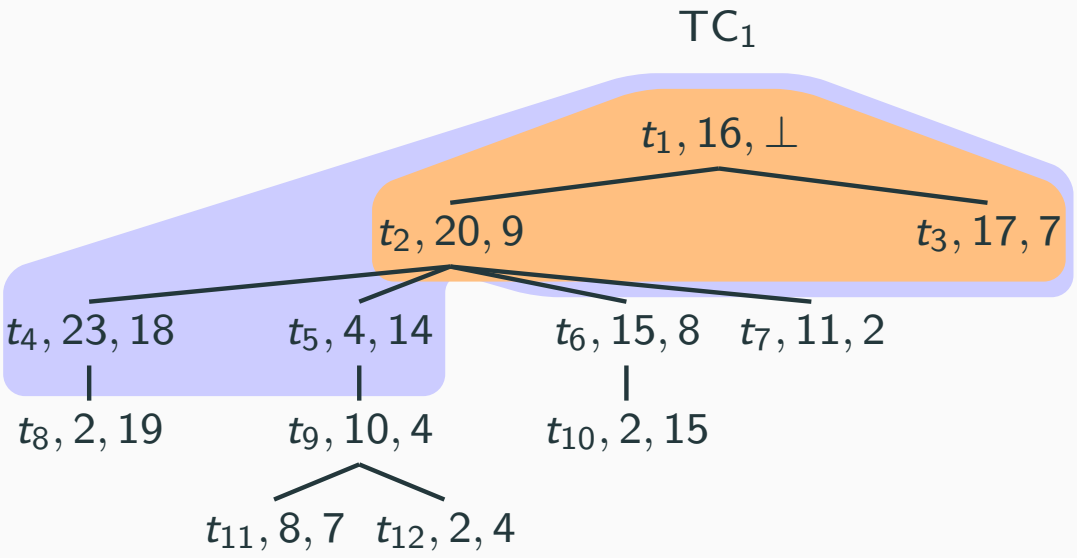


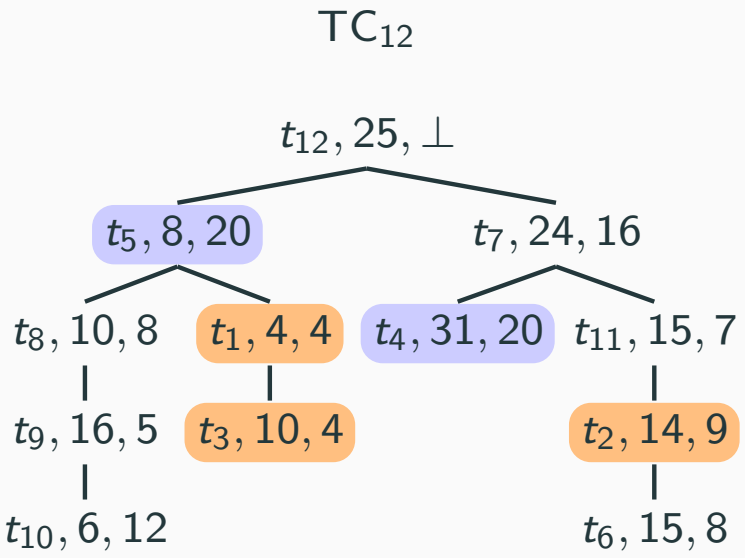
TC_{12} .Join(TC_1)



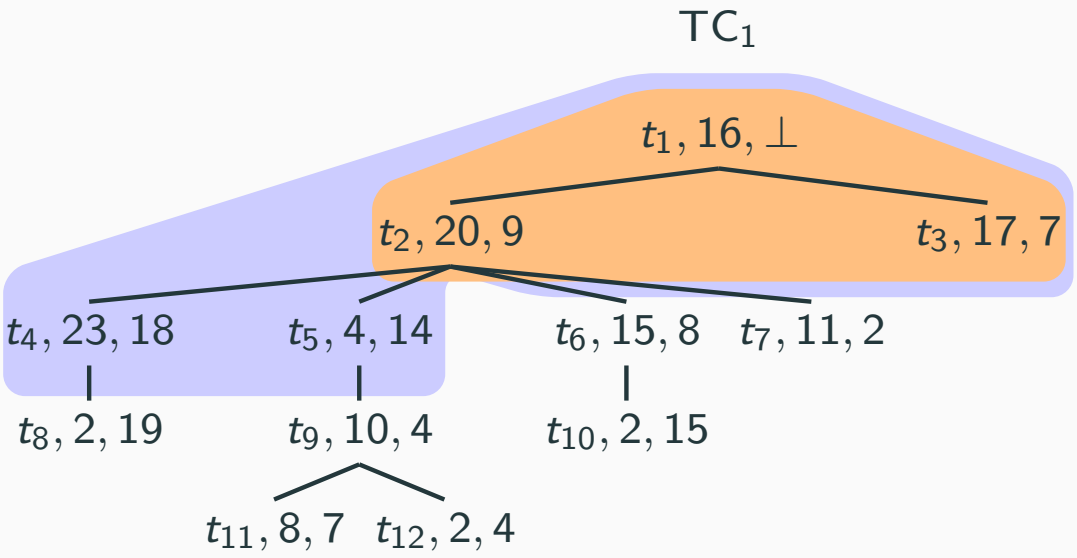


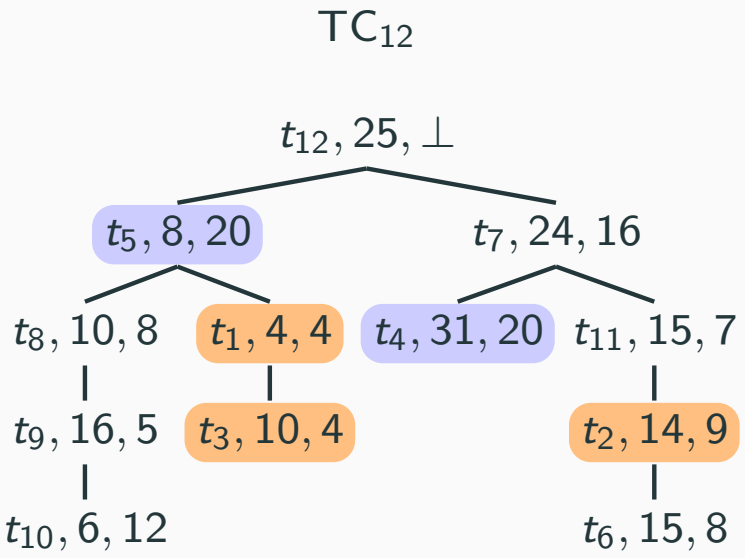
TC_{12} .Join(TC_1)

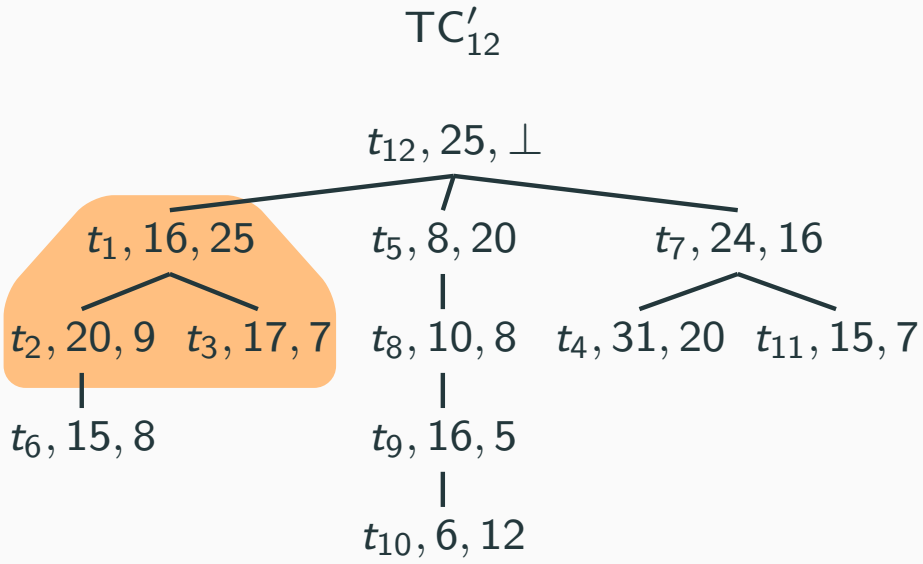




TC_{12} .Join(TC_1)







Copy?

In general, a tree clock copy will touch the whole tree, taking $O(k)$ time

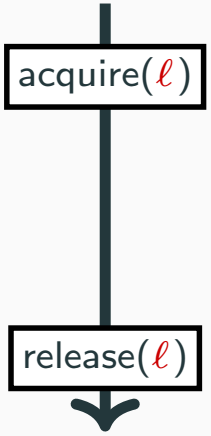
Copy?

In general, a tree clock copy will touch the whole tree, taking $O(k)$ time

But!

Algorithm: HB

```
procedure acquire(t, ℓ)
    C_t ← C_t ⊔ C_ℓ
procedure release(t, ℓ)
    C_ℓ = C_t
```



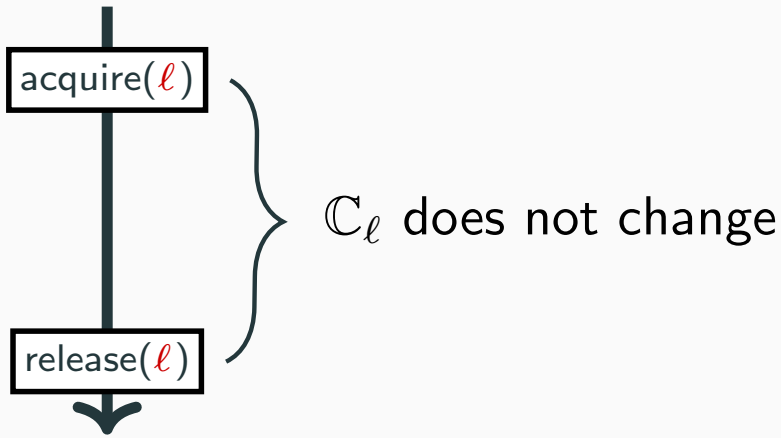
Copy?

In general, a tree clock copy will touch the whole tree, taking $O(k)$ time

But!

Algorithm: HB

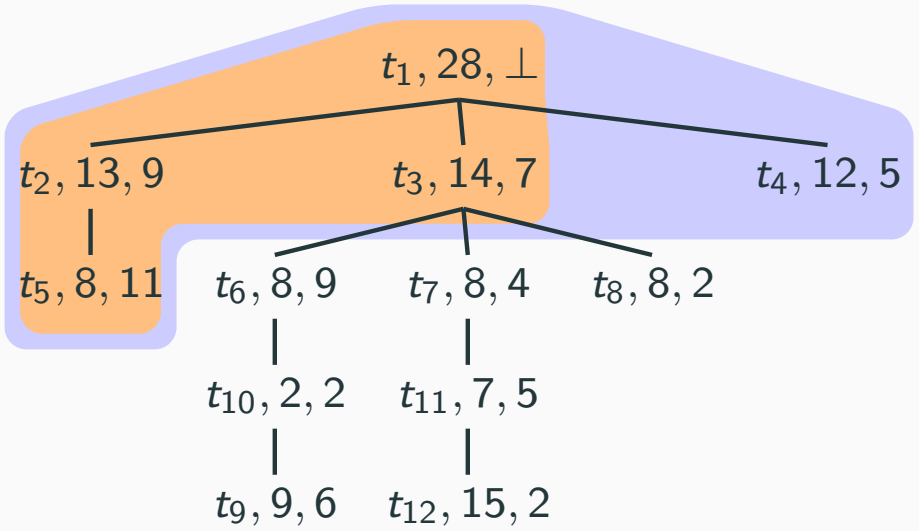
```
procedure acquire(t, ℓ)
    C_t ← C_t ⊔ C_ℓ
procedure release(t, ℓ)
    C_ℓ = C_t;    /* C_ℓ ⊑ C_t */
```



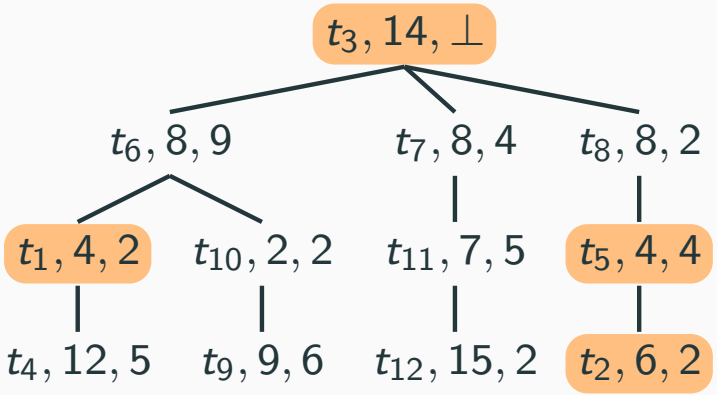
Copy is monotone $\rightarrow$ same semantics as Join!

TC_ℓ .MonotoneCopy(TC_1)

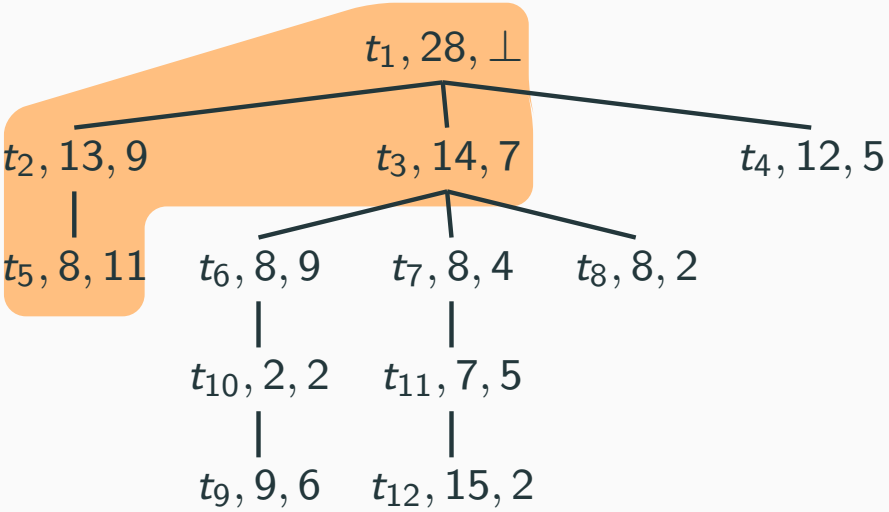
TC_1



TC_ℓ



TC'_ℓ



Tree clocks in Happens-Before

Algorithm: Happens-Before with **Vector Clocks**.

```
procedure acquire(t, ℓ)
    C_t.VectorClockJoin(C_ℓ)
procedure release(t, ℓ)
    C_ℓ.VectorClockCopy(C_t)
```

Algorithm: Happens-Before with **Tree Clocks**.

```
procedure acquire(t, ℓ)
    C_t.TreeClockJoin(C_ℓ)
procedure release(t, ℓ)
    C_ℓ.TreeClockCopy(C_t)
```

Data Structure Optimality for Happens-Before

What is the optimal data structure for Happens-Before?

Data Structure Optimality for Happens-Before

What is the optimal data structure for Happens-Before?

Tree clock optimality for Happens-Before

No other data structure can offer asymptotically better performance.

Data Structure Optimality for Happens-Before

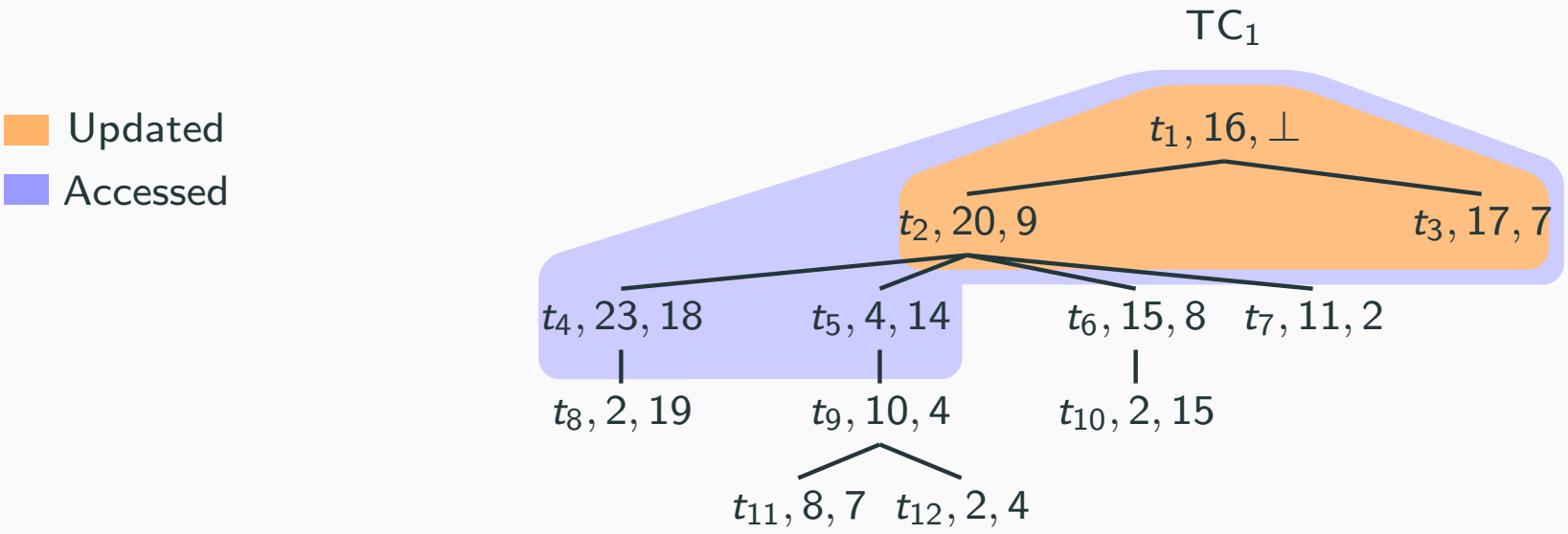
What is the optimal data structure for Happens-Before?

Tree clock optimality for Happens-Before

No other data structure can offer asymptotically better performance.

- Tree clocks perform at most 3 times more work than necessary.

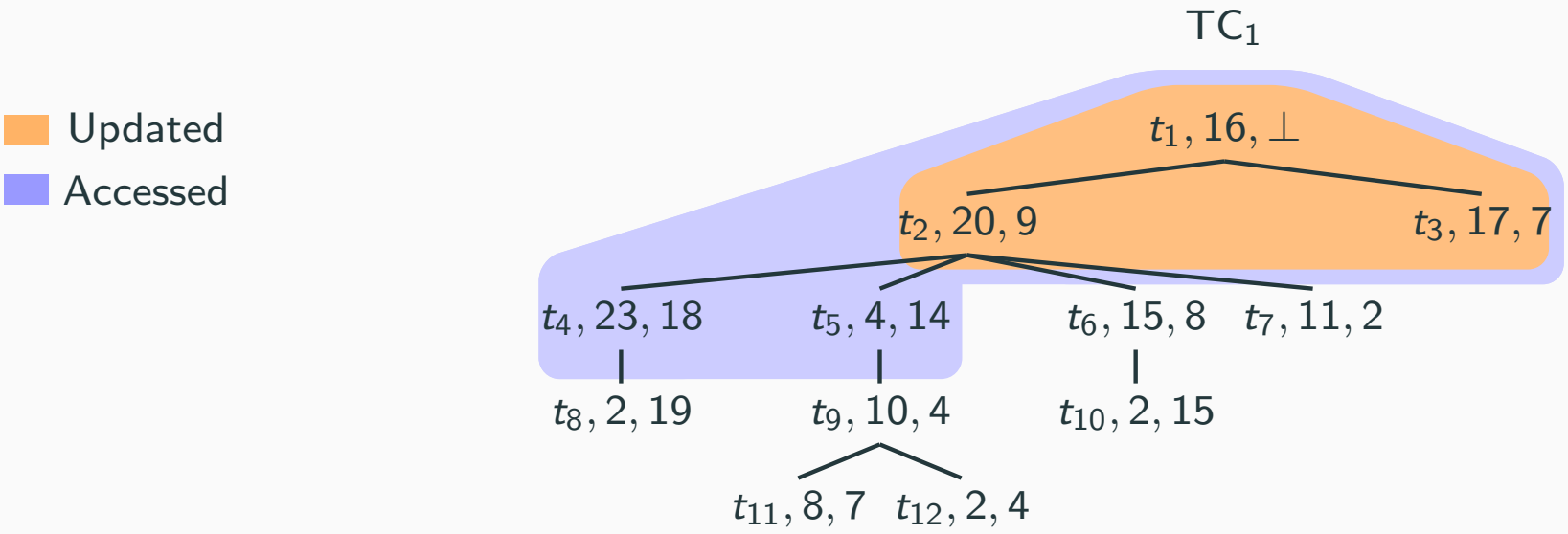
Data Structure Optimality for Happens-Before



Vector time work $\mathrm{VTWork}(\sigma)$

$\mathrm{VTWork}(\sigma)$ = the *smallest* number of data-structure accesses on σ

Data Structure Optimality for Happens-Before



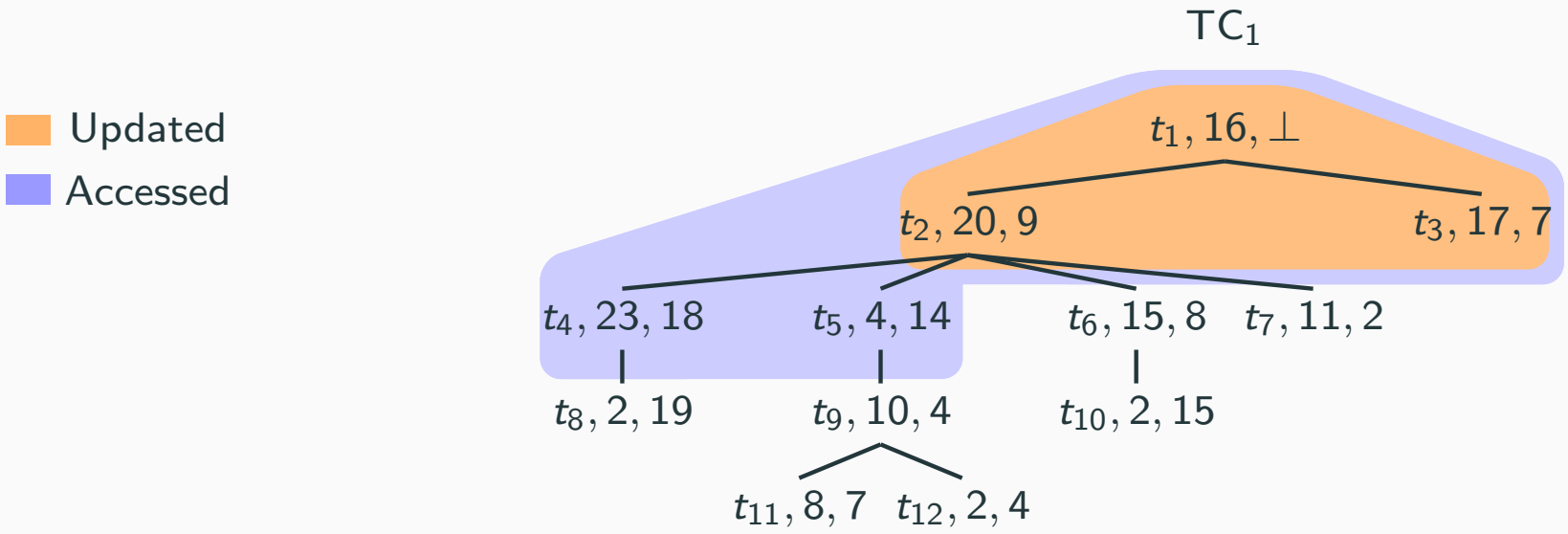
Vector time work $\mathrm{VTWork}(\sigma)$

$\mathrm{VTWork}(\sigma)$ = the *smallest* number of data-structure accesses on σ

Tree clock work $\mathrm{TCWork}(\sigma)$

$\mathrm{TCWork}(\sigma)$ = the total number of *tree* clock entries accessed on σ

Data Structure Optimality for Happens-Before



Vector time work $\mathrm{VTWork}(\sigma)$

$\mathrm{VTWork}(\sigma)$ = the *smallest* number of data-structure accesses on σ

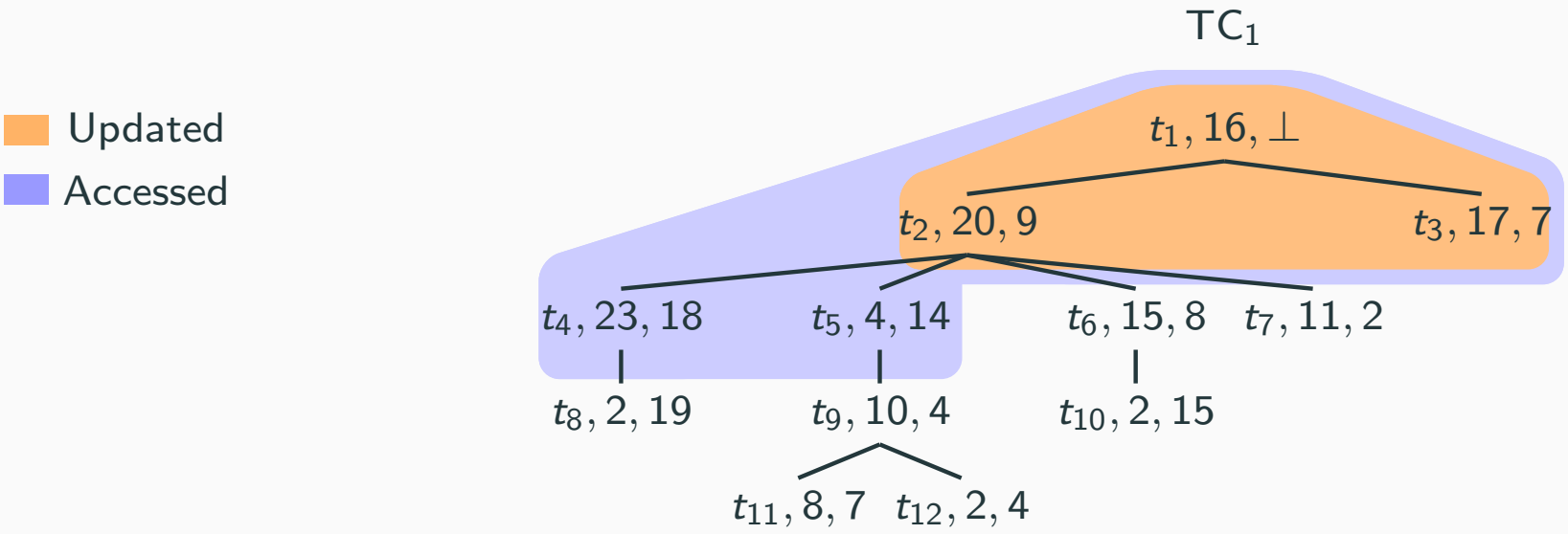
Tree clock work $\mathrm{TCWork}(\sigma)$

$\mathrm{TCWork}(\sigma)$ = the total number of *tree* clock entries accessed on σ

Vector clock work $\mathrm{VCWork}(\sigma)$

$\mathrm{VCWork}(\sigma)$ = the total number of *vector* clock entries accessed on σ

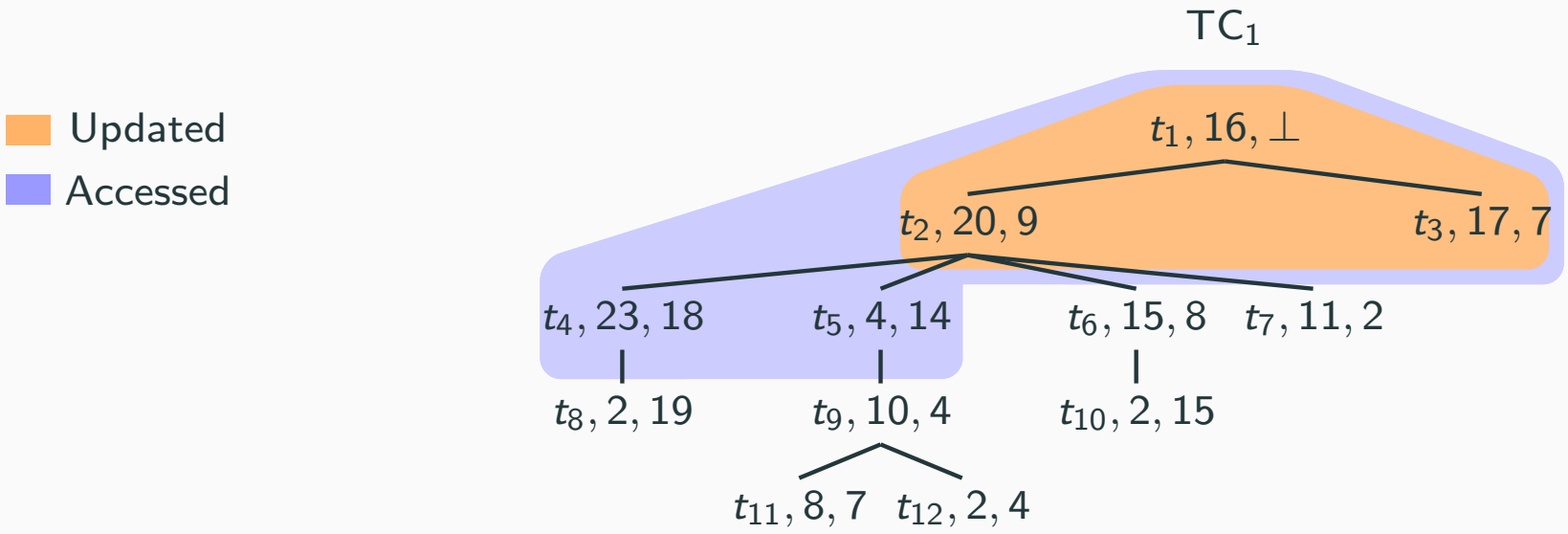
Data Structure Optimality for Happens-Before



$\text{VCWork}(\sigma) \leq k \cdot \text{VTWork}(\sigma)$

$\text{VCWork}(\sigma)$ can be k times worse

Data Structure Optimality for Happens-Before



$\mathsf{VCWork}(\sigma) \leq k \cdot \mathsf{VTWork}(\sigma)$

$\mathsf{VCWork}(\sigma)$ can be k times worse

Theorem

$\mathsf{TCWork}(\sigma) \leq 3 \cdot \mathsf{VTWork}(\sigma)$

Tree clocks are (asymptotically) VT-optimal!

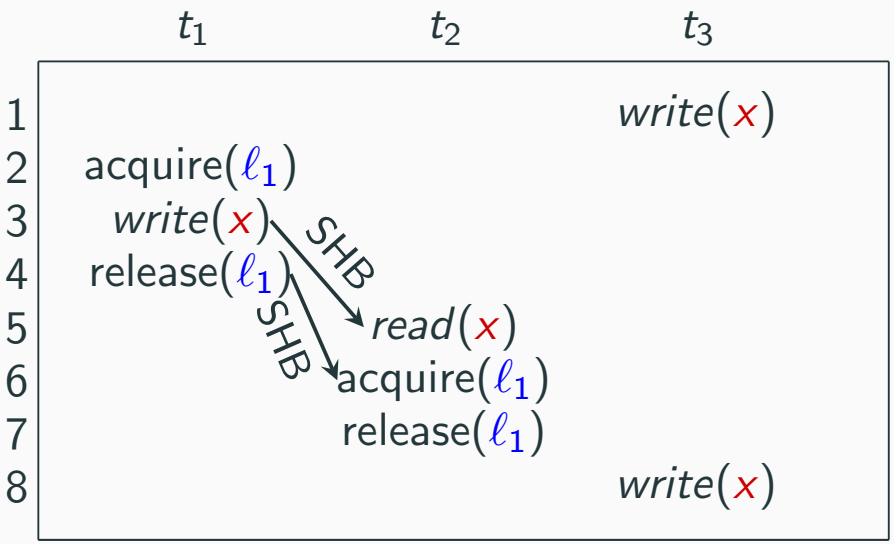
Beyond Happens-Before

- Tree clocks can be used in other partial orders

Beyond Happens-Before

- Tree clocks can be used in other partial orders

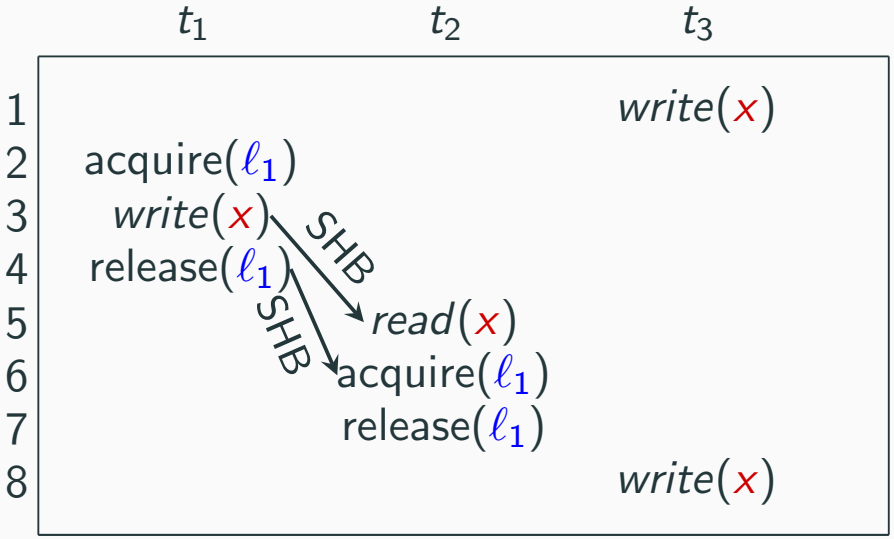
Schedulable-Happens-Before (SHB)



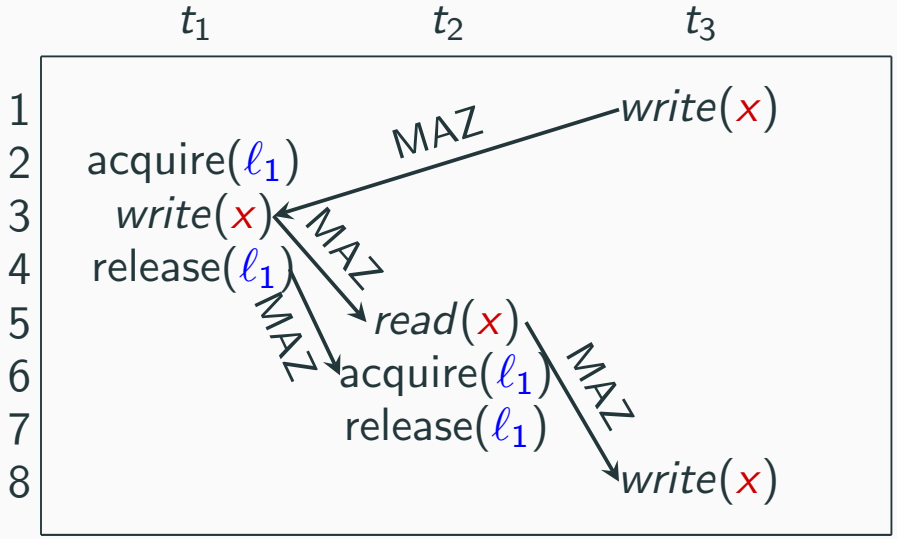
Beyond Happens-Before

- Tree clocks can be used in other partial orders

Schedulable-Happens-Before (SHB)



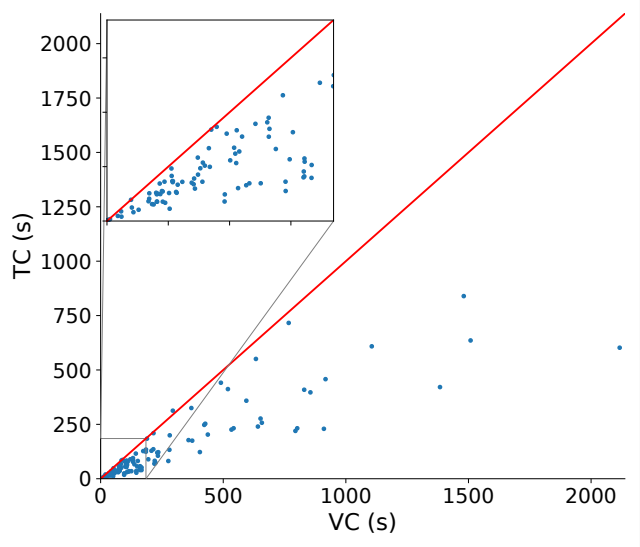
Mazurkiewicz (MAZ)

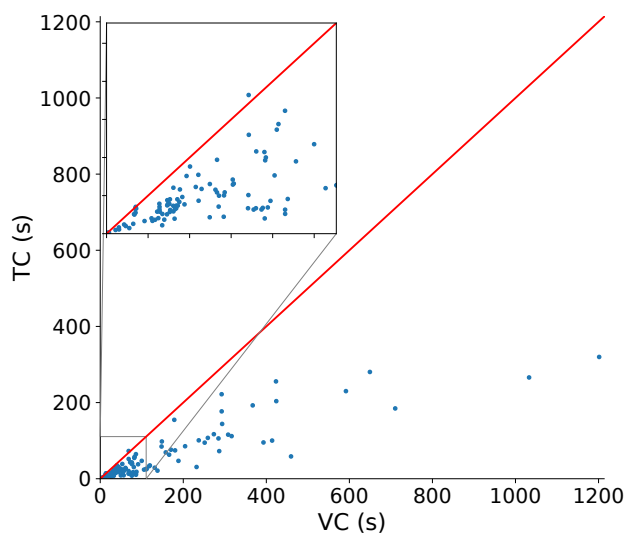


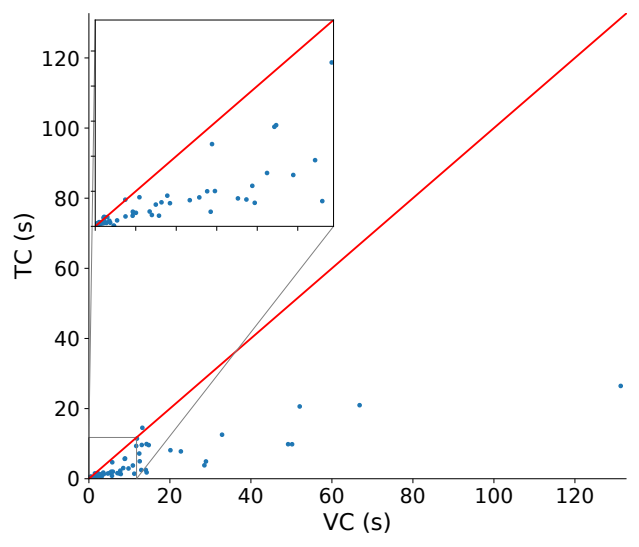
Some Experiments

- 153 benchmark traces
  - Based on standard Java and OpenMP benchmark suites.
- Implemented HB, SHB and MAZ with both tree clocks and vector clocks
- Measured the time in the following tasks:
  - Compute the partial order
  - Perform the race analysis

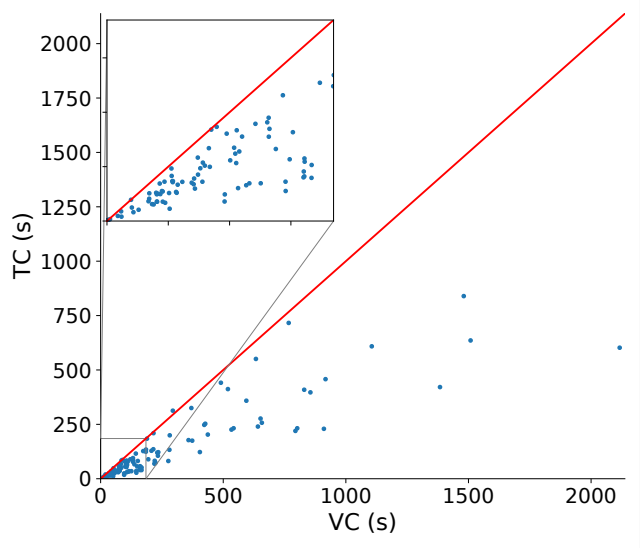
Times

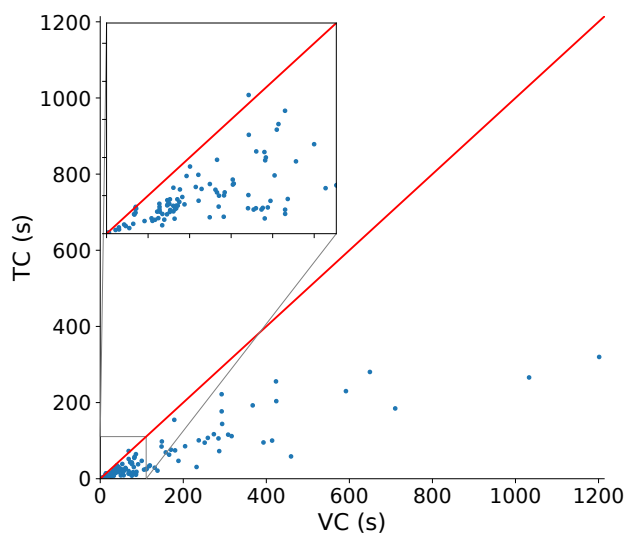

TC (s)
VC (s)
0
250
500
750
1000
1250
1500
1750
2000

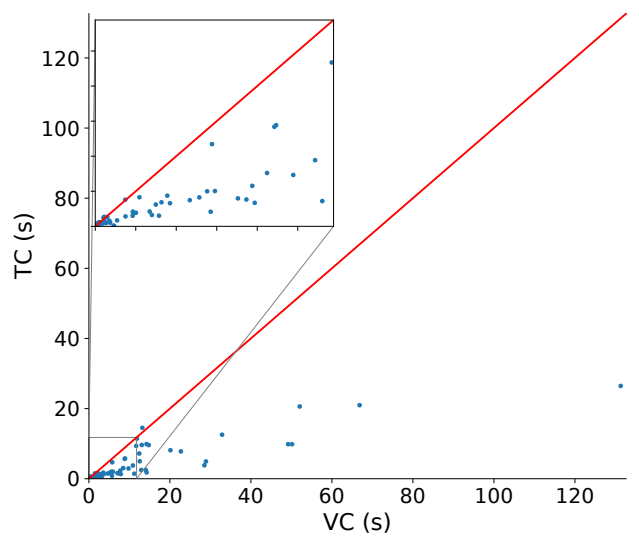

TC (s)
VC (s)
0
200
400
600
800
1000
1200


TC (s)
VC (s)
0
20
40
60
80
100
120

Times





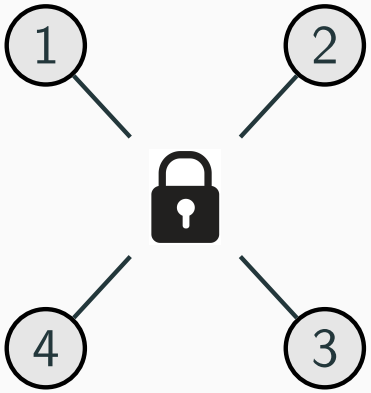


	Mazurkiewicz	*Schedulable-Happens-Before*	*Happens-Before*
Partial Order	2.02×	2.66×	2.97×
Partial Order + Analysis	1.49×	1.80×	1.11×

Scalability Tests

Controlled experiment: threads communicate over a single lock

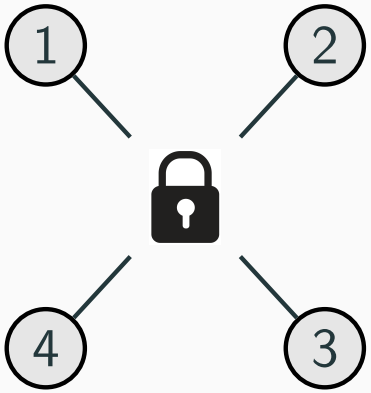
- Theoretical speedup: $\frac{2 \cdot k}{k/2+1} \simeq 4\times$

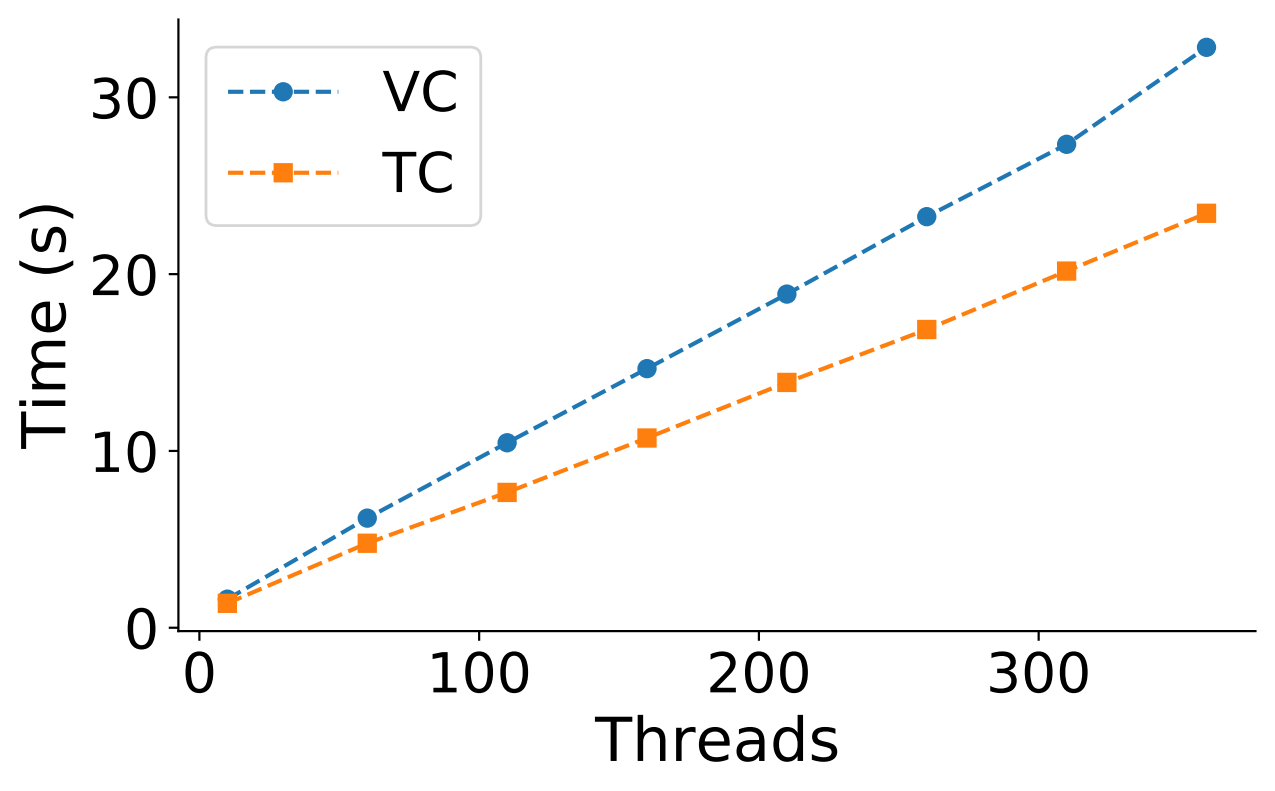


Scalability Tests

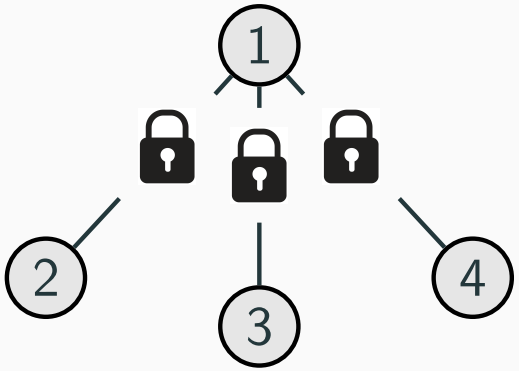
Controlled experiment: threads communicate over a single lock

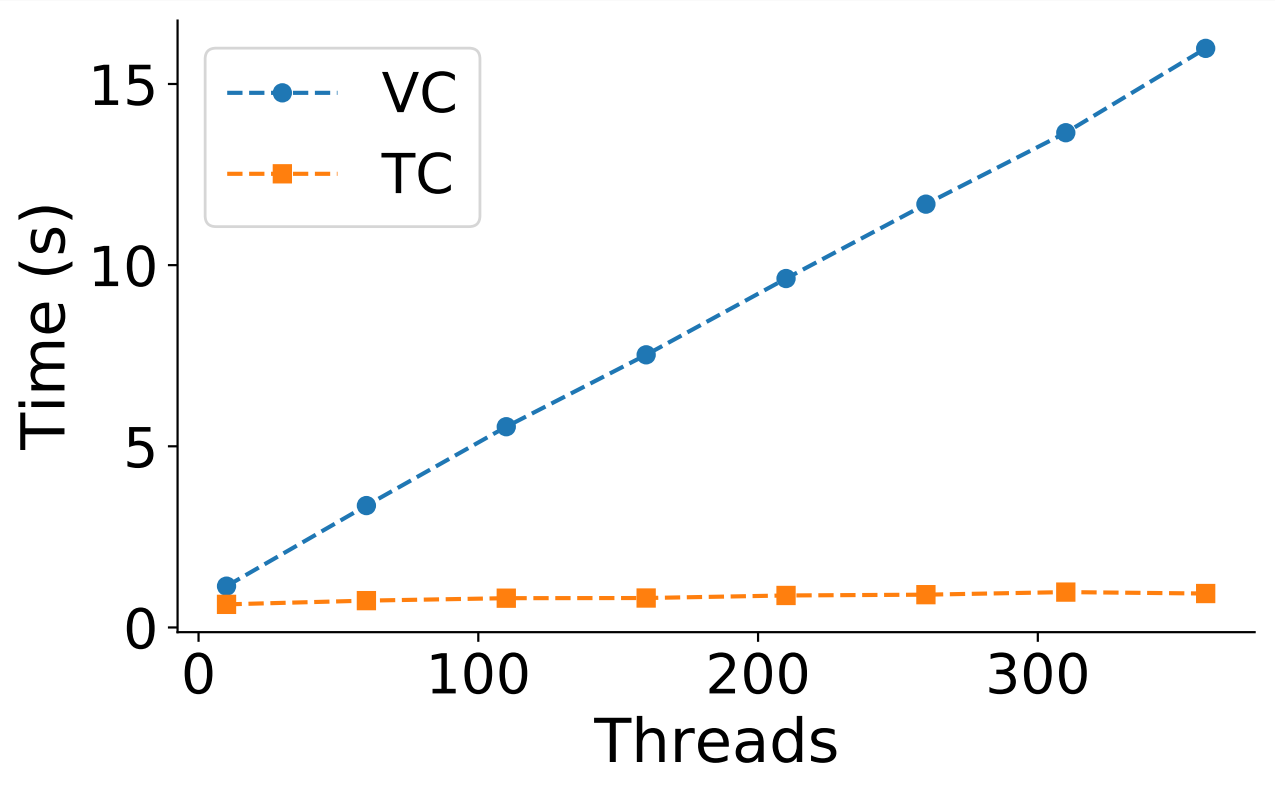
- Theoretical speedup: $\frac{2 \cdot k}{k/2+1} \simeq 4\times$
- Observed speedup: $1.33\times$



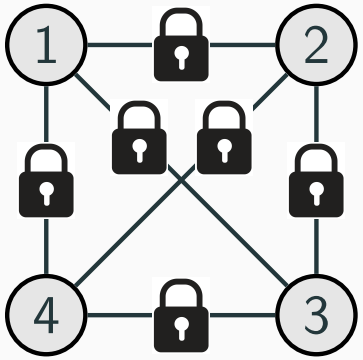


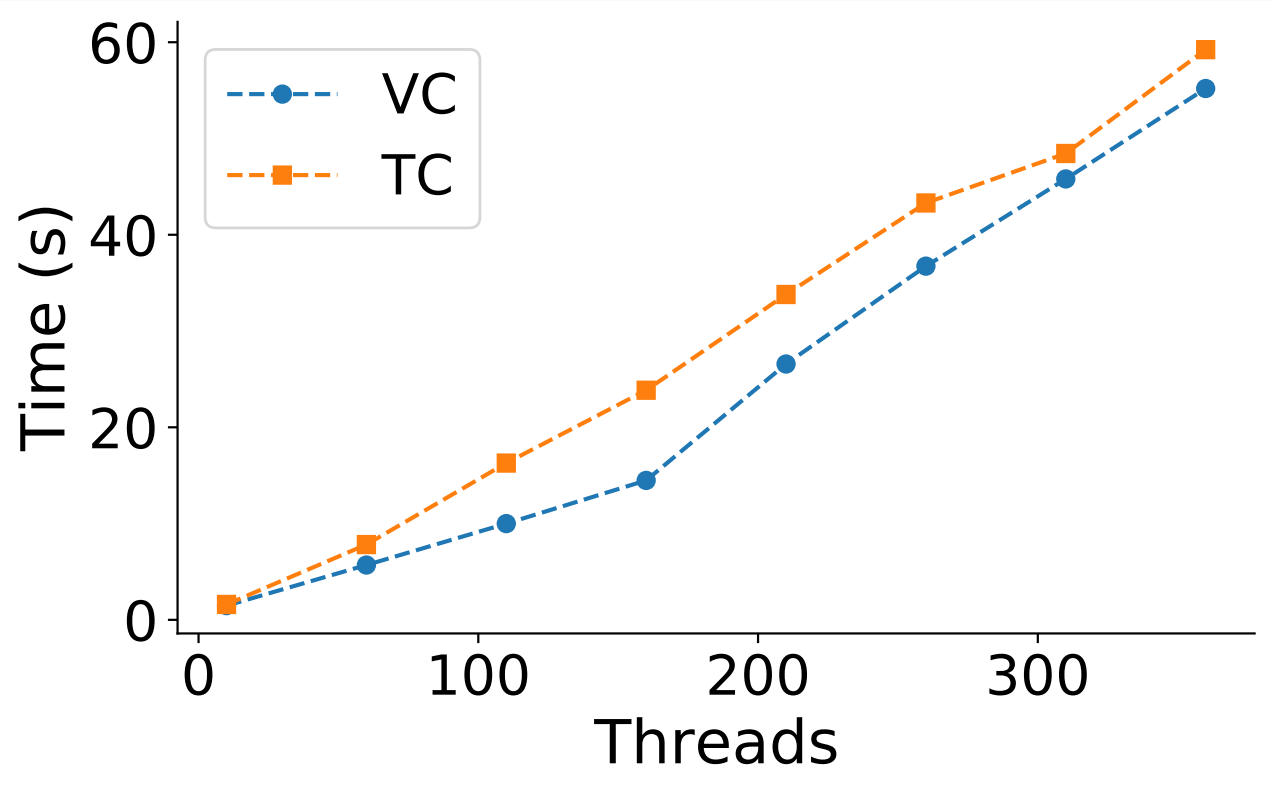
Scalability Tests





Scalability Tests

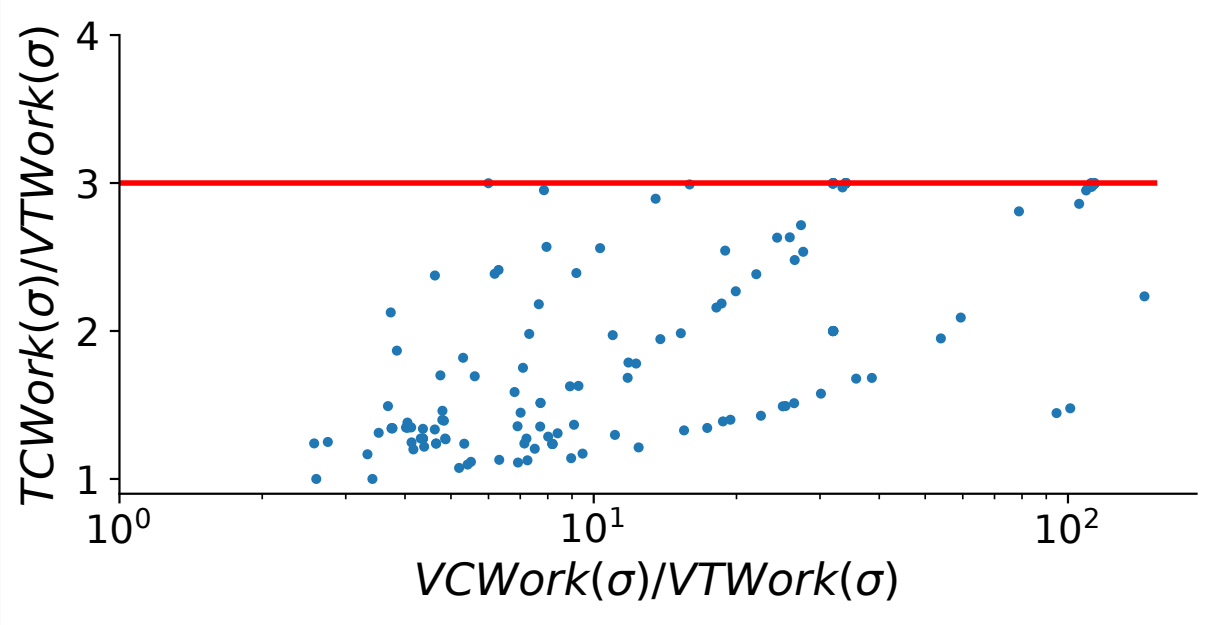




Experimental Results - Optimality for Happens-Before

Theorem

$\mathsf{TCWork}(\sigma) \leq 3 \cdot \mathsf{VTWork}(\sigma)$



Tree Clock Summary

// Initialize a tree clock for thread t **function** `Init`(t)	$O(1)$
// Get the clock for thread t **function** `Get`(t)	$O(1)$
// Increment the clock of the root thread **function** `Increment`(i)	$O(1)$
// True iff $\sqsubseteq \mathrm{TC}'$ **function** `LessThan`(TC')	$O(1)$
// Update with $\sqcup \mathrm{TC}'$ **function** `Join`(TC')	$O(k)$
// Monotone copy, assumes that this $\sqsubseteq \mathrm{TC}'$ **function** `MonotoneCopy`(TC')	$O(k)$
// Regular copy **function** `Copy`(TC')	$O(k)$

Tree Clock Summary

`// Initialize a tree clock for thread` t **function** `Init(`t`)`	$O(1)$	
`// Get the clock for thread` t **function** `Get(`t`)`	$O(1)$	
`// Increment the clock of the root thread` **function** `Increment(`i`)`	$O(1)$	
`// True iff` $\sqsubseteq \mathrm{TC}'$ **function** `LessThan(`TC'`)`	$O(1)$	
`// Update with` $\sqcup \mathrm{TC}'$ **function** `Join(`TC'`)`	$O(k)$	$\rightarrow$ **effectively sublinear**
`// Monotone copy, assumes that this` $\sqsubseteq \mathrm{TC}'$ **function** `MonotoneCopy(`TC'`)`	$O(k)$	$\rightarrow$ **effectively sublinear**
`// Regular copy` **function** `Copy(`TC'`)`	$O(k)$	

Tree Clock Summary

`// Initialize a tree clock for thread` t **function** `Init(`t`)`	$O(1)$	
`// Get the clock for thread` t **function** `Get(`t`)`	$O(1)$	
`// Increment the clock of the root thread` **function** `Increment(`i`)`	$O(1)$	
`// True iff` $\sqsubseteq \mathrm{TC}'$ **function** `LessThan(`TC'`)`	$O(1)$	
`// Update with` $\sqcup \mathrm{TC}'$ **function** `Join(`TC'`)`	$O(k)$	$\rightarrow$ **effectively sublinear**
`// Monotone copy, assumes that this` $\sqsubseteq \mathrm{TC}'$ **function** `MonotoneCopy(`TC'`)`	$O(k)$	$\rightarrow$ **effectively sublinear**
`// Regular copy` **function** `Copy(`TC'`)`	$O(k)$	

Thank you!